VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Hardwarově akcelované algoritmy zpracování obrazu na mobilní platformě Nvidia

Hardware Accelerated Image Processing Algorithms on Nvidia Mobile Platform

2014 Lukáš Kroczek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Lukáš Kroczek**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Hardwarově akcelované algoritmy zpracování obrazu na mobilní platformě NVIDIA
Hardware Accelerated Image Processing Algorithms on NVIDIA Mobile Platform

Zásady pro vypracování:

V současné době je mnoho algoritmů pro zpracování obrazu aplikováno na mobilních zařízeních. Jejich náročnost však mnohdy vyžaduje hardwarovou akceleraci. Některá zařízení jsou v současnoti vybavena produkty firmy NVIDIA, které mohou být využity k akceleraci takovýchto algoritmů. Cílem práce je implementace sady algoritmů pro zpracování obrazu a jejich akcelerovaných variant s otestováním na různých mobilních platformách.
Ve své práci proveďte:
1. Seznamte se s algoritmy pro hledaní vzorů pomocí hrubé síly, Haarova detektoru, LBP apod. ve zpracování obrazu.
2. Prostudujte možnosti urychlení algoritmů na Vámi dostupných mobilních platformách.
3. Implementujte vybrané algoritmy a otestujte jejich urychlení.
4. Dosažené výsledky řádně popište.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Jan Gaura**

Datum zadání: 16.11.2012

Datum odevzdání: 07.05.2014

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 1. května 2014 .....

Velice moc bych chtěl tady poděkovat vedoucímu mé práce panu Gaurovi za nápady a připomínky, vyučujícím za předání vědomostí, kolegům, rodičům a přítelkyni za podporu a dobrá slova.

Abstrakt

Cílem této diplomové práce je nastudovat a naprogramovat vybrané algoritmy zpracování obrazu a následně vytvořit jejich hardwarově akcelerovanou verzi, která poběží na mobilním zařízení s procesorem od společnosti Nvidia. Společnost Nvidia vyvinula pro mobilní zařízení procesory Tegra, které jsou obsaženy v několika prodávaných modelech tabletů. Algoritmy popsané a naprogramované v této práci budou testovány na zařízení Nexus 7. Vybrané algoritmy, které budu v práci hardwarově akcelerovat, jsou výpočty Integrálního obrazu, Lokální binární vzory a Haarovy příznaky.

Klíčová slova: Android, Dalvik, Java, Aplikace, Tegra 3, C++, jni, NEON, Zpracování obrazu, Integrální obraz, Detekce objektů, LBP, Haarovy příznaky

Abstract

Goal of this diploma thesis are study and implement selected image processing algoritms and create hardware accelerated version of this algoritms. Algoritms will work on mobile platform with Nvidia procesor. Company Nvidia create Tegra procesors for mobile devices which are used in few models of tablet devices. Algoritms subscribed and implemented in this diploma thesis will be tested on Nexus 7 device. Selected image processing algoritms which will be hardware accelerated are integral image, Local Binary patterns and Haar-like features.

Keywords: Android, Dalvik, Java, Application, Tegra 3, C++, jni, NEON, Image processing, Integral image, Object detection, LBP, Haar-like features

Seznam použitých zkratek a symbolů

API	–	Aplication Programming interface
CPU	–	Central processing unit
CUDA	–	Compute Device Unified Architecture
FPS	–	Frames Per Second
GPU	–	Graphic processing Unit
GPGPU	–	General Purpose Computing on Graphic Processing Unit
LBP	–	Local Binary patterns - Lokální binární vzory
OpenCV	–	Open source Computer vision
RGB	–	Red Green Blue
SIMD	–	Single instruction multiple data
SAD	–	Sum of absolute differences
VFP	–	Vector Floating Point

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Tato práce se zabývá hardwarovou akcelerací algoritmů, které jsou používány pro detekování objektů v obraze. V dnešní době se metody detekování objektů využívají v mnoha různých aplikacích, nejběžněji se setkáváme například s detekcí obličejů v digitálních fotoaparátech nebo kamerách.

Čím dál tím častěji tyto algoritmy najdou využití taky na mobilních zařízeních. A pro mobilní zařízení budeme tyto algoritmy implementovat a následně hardwarově akcelerovat.

Jako nejzákladnější metoda hledání obličeje může být brána metoda porovnání vzorů. Tato metoda je založená na principu, kdy máme vzor a hledáme místo, kde tento vzor nejlépe zapadne, tedy mezi vzorem a daným místem v obraze je nejmenší rozdíl. Touto metodou se budu zabývat hlavně z důvodu, že pro výpočet se používá vlastnosti takzvaného integrálního obrazu, který budu hardwarově akcelerovat.

Pokročilé metody detekce objektů používají klasifikátory na určení, jestli se v obraze nachází hledaný objekt, nebo ne. Tyto klasifikátory ale potřebují mít obraz převedený do určitého tvaru, nebo mít určité informace o obrazu.

Jednou z možností, kterou budu v této práci hardwarově akcelerovat je výpočet takzvaných Lokálních Binárních vzorů, kdy v obraze detekuje čáry, hrany a rohy.

Druhou možností je využití detektoru, který byl navržen Violou a Jonesem [7]. Tento detektor je tvořen klasifikačním algoritmem a generátorem Haarových příznaků. V mé práci se budu věnovat právě tomuto generátoru a možnostmi jeho hardwarové akcelerace na mobilním zařízení.

Práce je členěna tak, že v první kapitole se budu věnovat teorii k jednotlivým algoritmům, v druhé kapitole se podíváme na vlastnosti testovacího zařízení, na platformu, na které je postaven, operační systém, který na zařízení pracuje a na možnosti, jak pro dané zařízení hardwarově akcelerovat algoritmy. Ve čtvrté kapitole si popíšeme základní vlastnosti a některé chování testovací aplikace a v páté kapitole se budeme věnovat implementaci a hardwarovou akcelerací jednotlivých algoritmů.

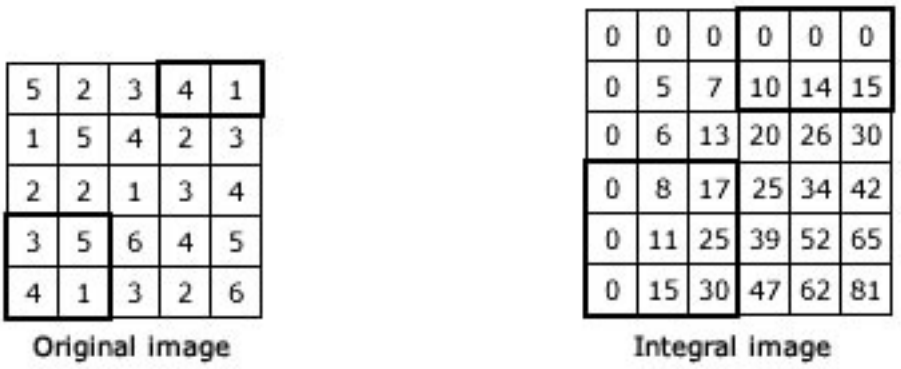
Cílem této práce není vytvořit aplikace pro detekci objektů, ale ukázat si možnosti, jak hardwarově akcelerovat tyto algoritmy na mobilní platformě Nvidia pomocí instrukcí NEON.

2 Vybrané algoritmy zpracování obrazu

V této kapitole si popíšeme teorii k algoritmům, které budu v rámci mé diplomové práce implementovat. Implementaci daných algoritmů najdete pak v kapitole 5.

2.1 Integrální obraz

Integrální obraz je způsob digitální reprezentace obrazu tak, že každý bod x představuje součet hodnot předchozích pixelů doleva a nahoru. Tedy pravý spodní bod obsahuje součet všech pixelů obrázku. Používá se k urychlení výpočtu, když v obrázku počítáme se sumaci částí obrazu.



Obrázek 1: Ukázka vypočítaného integrálního obrazu

Existují 2 typy integrálního obrazu, první má počet řádků a sloupců v integrálním obrazu větší o 1 oproti originálnímu obrazu. Je to způsobeno tím, že první řádek a první pixel na každém řádku má hodnotu 0. Druhá možnost integrální obraz má stejný počet řádků a sloupců jako originální obraz, ten nemá nulový první řádek a sloupec.

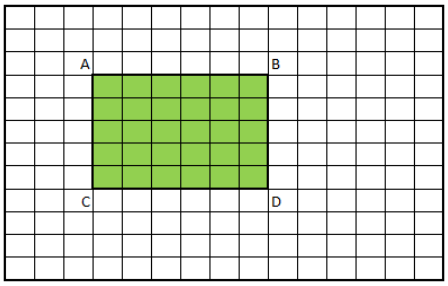
Rozhodl jsem se naimplementovat verzi, kde máme o jeden řádek a sloupec navíc z důvodu, že tento výpočet se bude následně lépe hardwarově akcelerovat, protože výpočet má méně podmínek.

Máme-li obraz i, hodnotu integrálního obrazu I na souřadnicích x,y vypočítáme podle vzorce [1]:

$$I(x,y) = \sum_{m=0}^{x} \sum_{n=0}^{y} i(m,n) \tag{1}$$

Vzhledem k tomu, že takový výpočet by byl neefektivní a několikanásobně by byly počítány sumy stejných oblastí, tak se využívá toho, že část integrálního obrazu je již vypočítána. Tím dojde ke zjednodušení náročnosti výpočtu a integrální obraz se pak vypočítá pomocí vzorce [1]:

$$I(x,y) = i(x,y) + I(x-1,y) + I(x,y-1) - I(x-1,y-1) \tag{2}$$



Obrázek 2: Ilustrace využití integrálního obrazu

V případě, že existuje vypočítaný integrální obraz, tak výpočet libovolně velké sumy pro obdélník zabere vždy pouze 4 přístupy do paměti a výpočet má pro jakoukoliv velikost konstantní čas výpočtu. Pro výpočet sumy hodnot pixelů zeleného obdélníku na obrázku 2 stačí znát souřadnice rohových bodů A, B, C a D. Celkovou sumu pro oblast ohraničenou 4 mi body lze vypočítat pomocí vzorce [1]:

$$\sum = I(A) + I(D) - I(C) - I(B) \tag{3}$$

Stejným způsobem lze počítat objemy i obdélníků pootočených o 45 stupňů. Tato vlastnost se využívá například při počítání pootočených Haarových příznaků [1].

2.2 Porovnání vzorů

Neboli anglicky Template Matching je technika zpracování digitálního obrazu pro nalezení malých částí obrazu, které korespondují nejlépe s vzorovým obrazem [2]. Tato technologie může být použita jako součást kontroly kvality, způsob, jak navigovat robota nebo jak detekovat hrany v obraze.

Tato technologie se nejlépe provádí na šedém obrázku, nebo na obrázku kde jsou zobrazeny pouze hrany.

Rozdíl mezi pixely se vyjádří pomocí SAD (Sum of absolute differences) hodnoty. V místě, kde bude toto SAD nejmenší je s největší pravděpodobností místo, kde se nachází objekt podobný našemu vzoru. Vzorec pro výpočet SAD je:

$$SAD(x,y) = \sum_{i=0}^{T_{rows}} \sum_{j=0}^{T_{cols}} Diff(x+i, y+j, i, j) \tag{4}$$

Procházení a hledání rozdílu na obrazu pro jednu smyčku si můžeme vyjádřit jako:

$$\sum_{x=0}^{S_{rows}} \sum_{y=0}^{S_{cols}} SAD(x,y) \tag{5}$$

V této technice ale vidím velké nedostatky. Hlavní nedostatek vidím to, že technika hledá pouze jeden a právě jeden výskyt, kde je SAD nejmenší. Není ošetřeno, co se stane

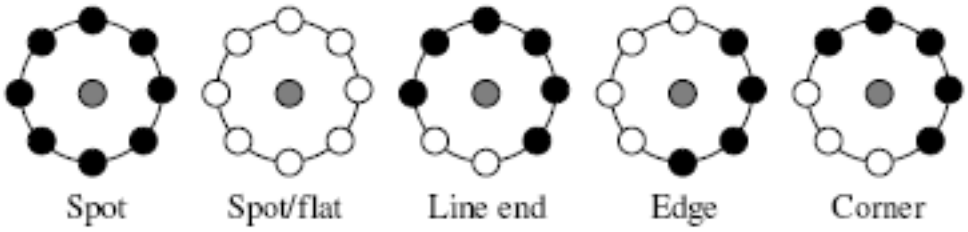
v případě, že nenajdeme podobný vzor ani z části, nebo v případě, kdy se vyskytne vícekrát. Jako další velké mínus vidím to, že by bylo velice časově náročné, kdybych měl porovnávat obraz pro více velikostí vzorů, takto máme jeden vzor tváře, ale tváře na obrazu můžou mít různou velikost. Další nevýhodou je to, že světelné podmínky můžou zásadně ovlivnit výsledek a algoritmus je na ně citlivý. Technika hledání vzorů je pro hledání obličeje velice nevhodná a vytvořená část kódu, která provádí porovnání vzorů, byla vytvořena pro to, abych mohl provést testování integrálního obrazu a aby šla vidět jeho funkčnost.

Na druhou stranu si myslím, že technika je vhodná k ukázkám, jak funguje integrální obraz a proč je vhodný v porovnání s tím, kdybychom integrální obraz neměli a počítali sumy při každém průchodu.

Ještě jedno důležité upozornění. Musíme si uvědomit, jaký typ integrálního obrazu používáme. V případě, že používáme Integrální obraz, který je větší o jeden pixel (v našem případě), tak nesmíme zapomenout ve všech případech přičíst k souřadnicím $x+1$ a $y+1$.

2.3 Lokální binární vzory

Anglicky Local binary patterns (používaná zkratka LBP), slouží k popisu vlastností obrázků pomocí příznaku, který obraz charakterizuje. V tomto případě je příznak reprezentován histogramem. Lokální binární vzor byl poprvé opsán v roce 1994[3][4]. Lokální binární vzor umožňuje texturní analýzu, která je prostorově invariantní a nezávislá na úrovni osvětlení. Je založena na rozpoznání jistých lokálních binárních vzorů (primitiv), které jsou homogenní a které popisují vlastnosti lokální textury. Nejčastějšími primitivy, které je možno detekovat metodou LBP jsou body, rohy a hrany [5][6].



Obrázek 3: Nejčastější primitiva, které je možno určit metodou LBP[5][6]

Při vytváření příznaku se prochází každý pixel obrázku (vyjímaje krajní pixely, protože nemá dostatek sousedních bodů) a přitom se pomocí ohodnocovací funkce získá jeho hodnota. Ohodnocovací funkce pracuje s okolními pixely, které jsou specifikovány pomocí parametru R, udávající vzdálenost bodů od aktuálního pixelu, a parametru P, který stanovuje počet bodů v daném okolí [5].

Vzorec pro výpočet hodnoty LPB pro různé varianty prvků P a R, kde g_c je centrální pixel[5].

$$LBP_{P,R} = \sum_{p=0}^{P-1} s(g_p - g_c)2^p \tag{6}$$

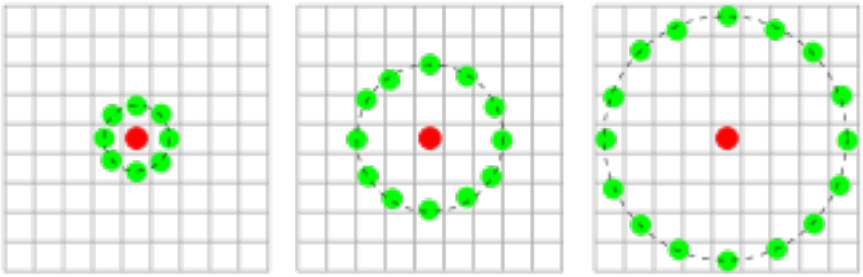
$$s(x) = \begin{cases} 1 & \text{pro } x >= 0 \\ 0 & \text{pro } x < 0 \end{cases} \tag{7}$$

Vstupní obraz se musí převést do stupně šedi (grayscale). Následně se začne procházet pixel po pixelu a kolem každého se uvažuje jeho definované okolí. Nad všemi prvky se provede prahování (thresholding), dále se výsledná hodnota vynásobí danou váhou a výsledek je LBP hodnota pro daný centrální bod (g_c) [5].

1	2	4
128	g_c	8
64	32	16

Tabulka 1: Ukázka vah okolních bodů pro R=1 a P=8

Ze všech LPB hodnot je následně vytvořen histogram četnosti výskytu, tzn. příznak obrázku. Tento výpočet je základní variantou LBP metody. Její názorná ukázka je na obrázku.



Obrázek 4: Tři příklady okolí použité jako definice okolí a pro spočítání lokálního binárního vzoru (local binary pattern, LBP). Na prvním obrázku jsou hodnoty $R = 1$ a $P = 8$, na druhém obrázku je $R = 2$ a $P = 12$ a na třetím obrázku jsou hodnoty $R = 4$ a $P = 16$ [5]

U vzorce pro výpočet LBP jsem provedl ještě dílčí změnu, která neovlivní výsledek, ale odstraní mi jeden krok výpočtu (odečítání hodnot). V originálním vzorci je vytvořen rozdíl mezi g_c a g_p a následně je tento rozdíl porovnáván, jestli je menší roven nebo menší, než 0. V upraveném vzorci jsem si zavedl funkci l, která má 2 vstupy a v případě, že g_p je větší, než g_c, tak funkce vrátí 1, naopak vrátí 0. Upravený vzorec vypadá následovně:

$$LBP_{P,R} = \sum_{p=0}^{P-1} l(g_p, g_c)2^p \tag{8}$$

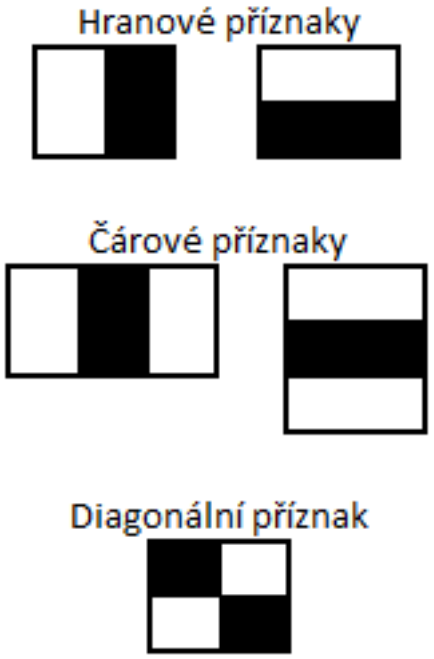
$$l(g_p, g_c) = \begin{cases} 1 & \text{pro } g_p >= g_c \\ 0 & \text{pro } g_p < g_c \end{cases} \tag{9}$$

2.4 Haarovy příznaky

Haarovy příznaky se používají k detekci objektů na digitálním obraze. Hodnota jednotlivého příznaku F_{haar} se počítá jako suma pixelu tmavé části příznaku R_{black} odečtená od sumy pixelů světlé části příznaku R_{white}.

$$F_{haar} = E(R_{white}) - E(R_{black}) \tag{10}$$

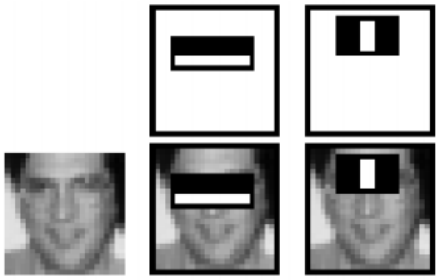
Jednotlivé příznaky se mohou skládat ze dvou (hranový příznak), tří (čárový příznak) nebo čtyř (diagonální příznak) obdélníkových oblastí. Haarovy příznaky uvedené na obrázku 5 patří k základní sadě příznaků, později se začala používat rozšířená sada příznaků, která obsahovala příznaky natočené o 45 stupňů [7].



Obrázek 5: Základní typy Haarových příznaků [7]

Hodnoty jednotlivých příznaků se počítají od minimální velikosti (například 2x2 pixelů) až do velikosti vstupního obrazu. K urychlení výpočtu Haarových příznaků se využívá integrálního obrazu.

Při detekci objektů na obrazu se využívá principu, kdy na obraze pohybuje podokno a v něm se provádí ohodnocení příznaky tak dlouho, dokud příznaky nejsou větší, než podokno. Podoknem se pohybuje v daném obrazu (zde se nejčastěji používá krok 2 pixely) tak dlouho, dokud není vypočítáno ohodnocení pro celý obraz. Následně se



Obrázek 6: Ukázka dvou typických pozic hledaných klasifikačním algoritmem na podokně pro Haarovy příznaky [7]

velikost podokna zvětšuje od minimální velikosti (většinou 24x24 pixelů) až do velikosti celého obrazu (například krokem 15%). Ohodnocení pro tyto podokna jsou posílány na ohodnocení klasifikačnímu algoritmu, který rozhodne, jestli se v daném podokně nachází nebo nenachází objekt, který se snažíme detekovat a na jehož detekování byl klasifikační algoritmus naučen.

V praxi je potřeba odezvu příznaku normalizovat kvůli změnám jasu v obrazu a změně velikosti příznaku [8]. Výpočet normalizované hodnoty Haarového příznaku, kde R_μ je suma všech bodů aktuálně počítaného podokna:

$$F_{haar} = \frac{E(F_{white}) - E(F_{black})}{\sqrt{\left|E(R_\mu)^2 - E(R_\mu^2)\right|}} \tag{11}$$

Všechny příznaky, jejich hodnoty, pozice a velikost jsou vstupem učícího klasifikačního algoritmu (například AdaBoost). Ten z nich potom vybere jen určité malé množství příznaků společně se stejným počtem natrénovaných slabých klasifikátorů, pomocí nichž lze vstupní obraz vhodně klasifikovat (stejný příznak se ve výsledném silném klasifikátoru může vyskytovat i několikrát, ovšem pokaždé s jiným nastavením slabého klasifikátoru). Pouze toto malé množství příznaků je pak použito při samotné detekci [7].

3 Testovací zařízení s mobilní platformou Nvidia

V této kapitole popíšu testovací zařízení, které jsem si vybral. Jako testovací zařízení postavené na platformě Nvidia jsem si vybral Nexus 7, který je vybavený procesorem Nvidia Tegra 3. Taky si v této kapitole popíšeme operační systém běžící na daném zařízení, tedy OS Android a možnosti tvorby aplikací pro tento operační systém.

Taky si v této kapitole popíšeme možnosti, jak hardwarově akcelerovat výpočty na platformě Nvidia Tegra. Na konci kapitoly si popíšeme knihovnu OpenCV, kterou budeme využívat v různých částech aplikace, přesněji podpůrné funkce z ní.

3.1 Nexus 7

Google Nexus 7 je první tablet, který si nechala firma Google vyrobit. Tento 7"tablet byl představen v červnu 2012 na Google I/O. Zároveň je tento tablet prvním čtyř jádrovým 7 palcovým tabletem na světě. Je vybaven technologiemi jako Wifi, Bluetooth, NFC. Ze senzorů tablet obsahuje GPS, gyroskop, akcelerometr, magnetometr a senzor přiblížení. Tablet obsahuje pouze jednu kameru a to přední s rozlišením 1280x1024. Tablet je osazen procesorem Nvidia Tegra 3 (model T30L 1.2 GHz na jádro, více informací v kapitole 3.2.2) a má 1 GB RAM DDR3L běžící na frekvenci 667MHz. Zařízení si společnost Google nechala vyrobit u společnosti Asus.

Aktuální verze systému je Android 4.4 (KitKat). Testovací zařízení obsahuje nejaktuálnější vydání určené pro daný model a to verzi 4.4.2.

3.2 Nvidia Tegra

Čipsety Tegra jsou takzvané "počítače na čipu"(Soc) vyvinuté společností Nvidia používající procesory ARM pro mobilní zařízení. Čipset Tegra integruje CPU, GPU, severní můstek, jižní můstek, řadič paměti a další čipy do jednoho balení.

Obrázek 7: Čipy Nvidia Tegra T20 (Tegra 2) and T30 (Tegra 3).

První představení procesorů Tegra proběhlo 12. února 2008[9], kdy byly přestaveny čipsety první generace. Tyto čipsety byly vydány ve dvou sériích, kdy série APX byla

určená pro využití ve smartphonech a série 600 byla určena pro použití v smartboocích a MID zařízeních. Na kongresu v roce 2009 představila společnost Nvidia port Android pro Tegru.

Dne 7. ledna 2010 společnost Nvidia představila Tegru 2, kdy hlavním podporovaných systémem byl Android. Na procesoru Tegra 2 jsou spustitelné i jiné operační systémy podporující architekturu ARM, pokud je pro ně dostupný bootloader. Takto například Tegra 2 podporuje Ubuntu GNU/Linux distribuci, která je dostupná na webu vývojářského fóra Nvidia[10].

V únoru 2011 Nvidia představila první 4 jádrový SoC. Tento čipset měl kódové označení Kal-El (pozn. autora: Kal-El je jméno, které dostal Clark Kent alias superman od svých rodičů na planetě Krypton předtím, než byla zničena), později byl tento čipset přejmenován na Tegra 3. Tento procesor byl použit na mnoha tabletech, které vyšly v druhé polovině roku 2011 a první polovině roku 2012, mezi nimi byl i Nexus 7.

V lednu 2012 Nvidia oznámila, že si automobilový výrobce Audi vybral jejich čip Tegra 3 pro zabudované informační a zobrazovací zařízení montovaná v automobilech [11].

6. ledna 2013 byl představen 4 jádrový procesor Tegra 4 s kódovým označením Wayne, který jako doprovodné jádro využívá nízkonapěťový Cortex A15. Velký pokrok nastal u počtu jader GPU, kdy Tegra 4 má 12x více jader oproti Tegře 3, tedy 72 jader. Tegra 4 byla dodávaná i ve variantě s označením Tegra 4i, kdy GPU obsahovala jenom 60 jader a doprovodné jádro bylo Cortex A9 [12][13].

3.2.1 Nvidia Tegra 2

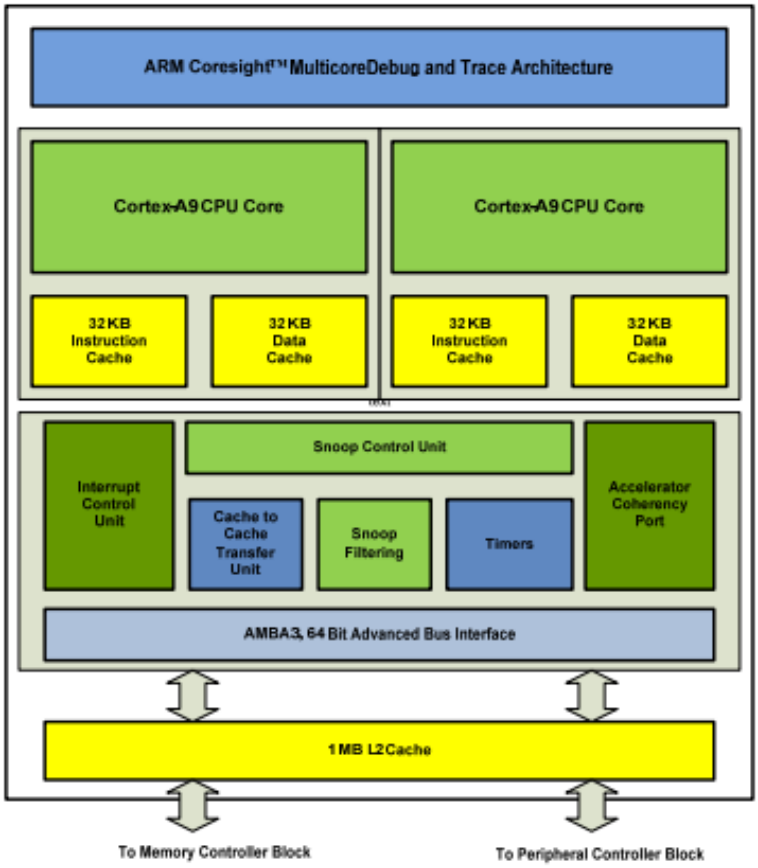
Druhá generace Tegry Soc byla osazena 2 jádrovým ARM Cortex-A9 procesorem, obsahovala ultra nízkonapěťové GPU s 8mi jádry. Čip byl vyráběn 40nm technologií. Jednalo se o první dvoujádrové CPU [14].

Tegra 2 podporuje hardwarovou akceleraci pro výpočty s plovoucí desetinnou čárkou, tzn. VFP (více v kapitole 3.3.2) ve verzi VFPv3-D16[10] a i když procesor je postaven na architektuře ARMv7 a má její instrukční sadu, tak chybí podpora ARM Advanced SIMD extension - NEON[10].

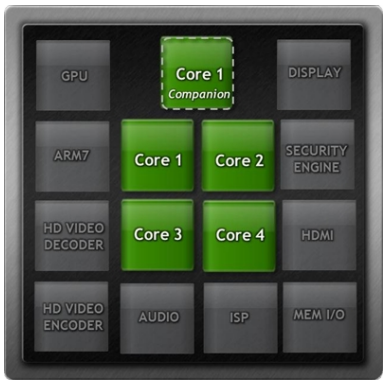
3.2.2 Nvidia Tegra 3

Tegra 3 je 4 jádrový procesor, jehož jádra jsou postaveny na architektuře ARM Cortex-A9 MPCore s ARMv7 instrukční sadou, který navíc obsahuje páté doprovodné jádro. Čip byl vyráběn 40nm technologií.

Doprovodné jádro je vyrobeno tak, aby umožňovalo běh na nízkých rychlostech a mělo tedy nízkou spotřebu. V případě potřeby výkonu se zapnou hlavní jádra (zapíná se jen tolik hlavních jader, kolik je aktuálně potřeba). Protože jádra jsou navržena tak, že nemůže běžet najednou doprovodné jádro a jakékoliv hlavní jádro, tak byla vyvinuta speciální logika pro velmi rychlé přepnutí mezi doprovodným jádrem a hlavními jádry. Tímto způsoben je zajištěno, že v případě, kdy zařízení není využíváno má co nejmenší spotřebu, ale v případě nutnosti má dostatečný výkon[15].

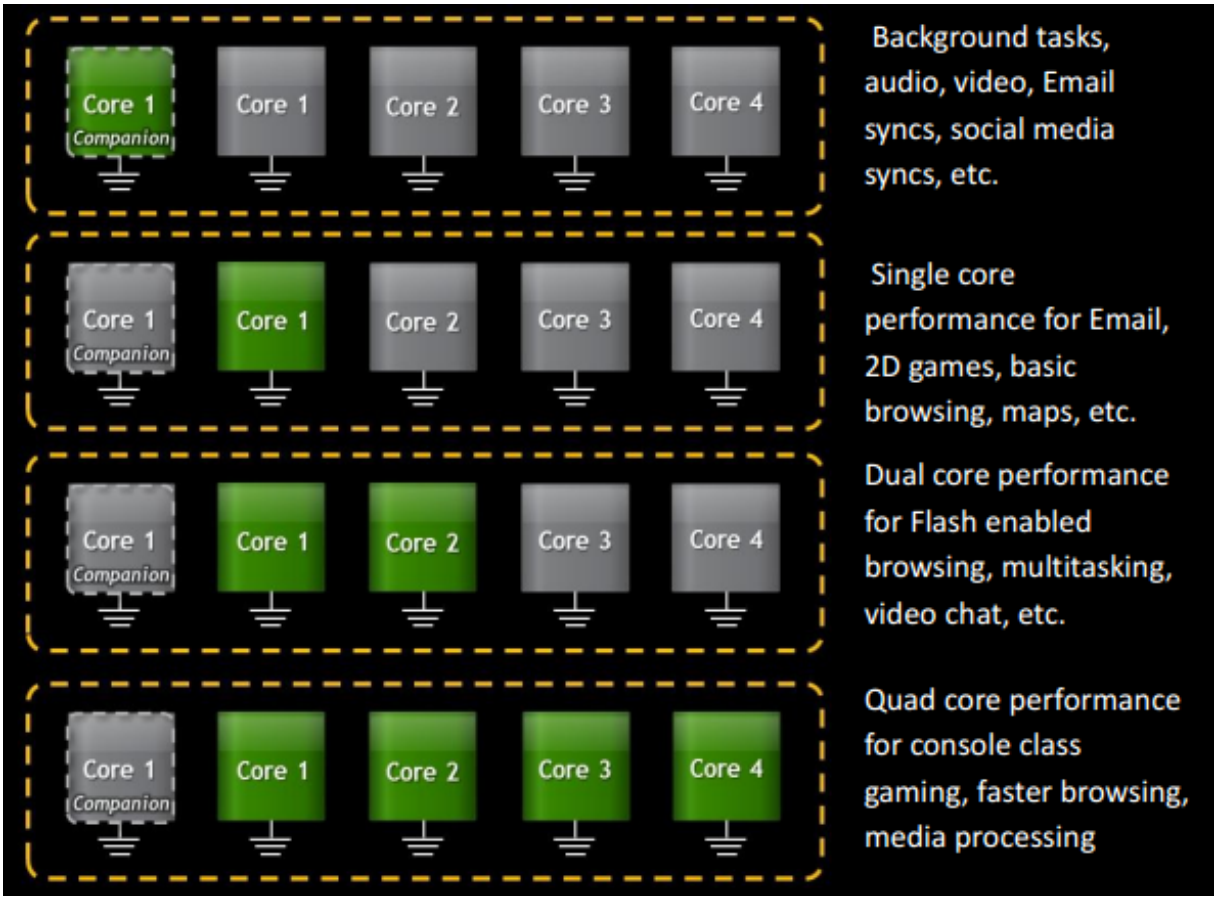


Obrázek 8: Implementace Tegra 2 2 jádrového ARM Cortex A9 MPcore procesoru[10].



Obrázek 9: Čipy v čipsetu Tegra 3 [15].

Tegra 3 je osazen 12ti jádrovým GPU s vysokou frekvencí. Tegra 3 podporuje hardwarovou akceleraci pro výpočty s plovoucí desetinnou čárkou, tzn. VFP ve verzi VFPv3 a taky je implementovaná instrukční sada ARM Advanced SIMD rozšíření - NEON[15]



Obrázek 10: Vysvětlení při jakých operacích se zapínají jádra. Procesor nikdy nemůže běžet tak, že by bylo najednou zapnuto doprovodné jádro a k tomu nějaké hlavní jádro [15].

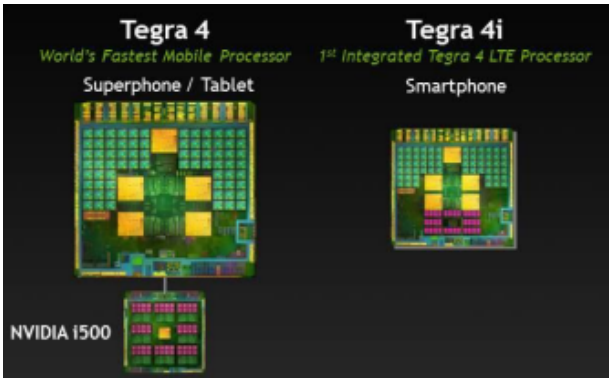
(více v kapitole 3.3.3).

Tegra 3 taky obsahuje architekturu Nvidia Direct Touch, která vylepšuje odezvu a snižuje nároky na napájení pro zpracování dotyků. Tegra 3D taky jako první z rodiny Tegra začala podporovat 3D stereo zvuk [16].

3.2.3 Nvidia Tegra 4

Procesor Tegra 4 je 4 jádrový procesor, jehož jádra jsou postavena na technologii Cortex A15 a má páté doprovodné jádro taky postavení na technologii Cortex A15. Všechny jádra mají ARMv7 instrukční sadu. GPU obsahuje 6x více jader oproti Tegra 3, tedy 72 jader. Čipy jsou jako první z rodiny Tegra vyráběny 28nm technologií [12][13].

Tegra 4i je levnější varianta Tegra 4, která se liší hlavně v tom, že místo Cortex A15 jsou použity Cortex A9 (jak u hlavních tak i u doprovodného jádra). Čp je taky vyráběn 28



Obrázek 11: Schéma jader pro procesory Nvidia Tegra 4 a Tegra 4i [13]

nm technologií a GPU obsahuje 60 jader [12][13]. Tegra 4i vyniká tím, že v sobě integruje LTE modem [13].

3.2.4 Nvidia Tegra K1

Plánovaná verze, která by měla mít GPU se 192 jádry s architekturou Kepler. Vyráběna by měla být 28 nm technologií. Tegra K1 má vzniknout i v 64 bitové variantě [17]. Výhodou Tegra K1 má být podpora technologie CUDA pro paralelizaci výpočtů na GPU. U Tegra K1 má dojít ke sjednocení vývoje, architektury a instrukčních sad u Tegra a GeForce [17][18].

3.3 Možnosti hardwarové akcelerace pro Nvidia Tegra 3

Jádra procesoru obsažené v procesoru Nvidia Tegra 3 jsou postavená na architektuře ARMv7-A. Tato architektura pro hardwarové zpracování nabízí technologie VFPv3, Thumb 2 a Advanced SIMD extension - NEON.

Mezi technologie pro hardwarové zpracování, které procesor Tegra 3 nepodporuje, patří CUDA.

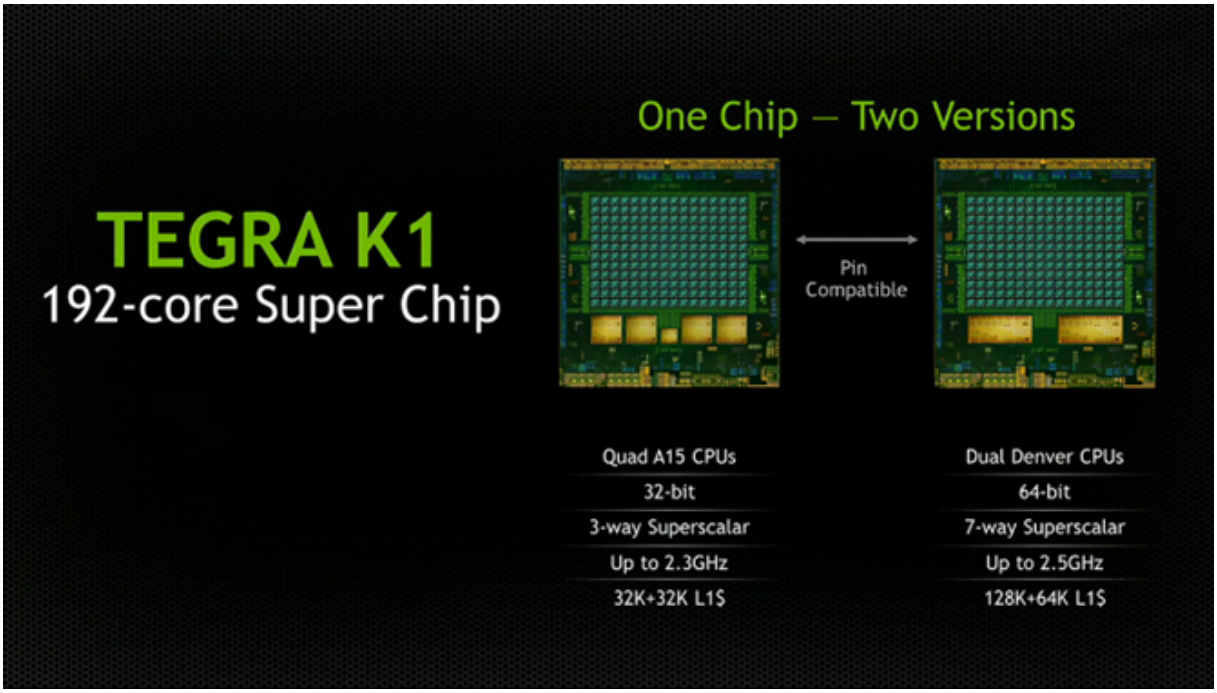
3.3.1 Thumb-2

Technologie Thumb2 byla představena v roce 2003. Thumb 2 rozšiřuje omezený 16-bitovou instrukční sadu u technologie Thumb a přidává 32-bitovou podporu, aby instrukční sada byla více univerzální.

Všechny čipy postavené na ARMv7 podporují Thumb2 instrukční sadu. Dokonce Cortex-M podporuje pouze Thumb instruční sadu [19][20].

3.3.2 Floating-point (VFP)

Vector Floating Point (VFP) je rozšíření technologie koprocesoru FPU na architektuře ARM. VFP poskytuje výpočty s desetinnou čárkou vhodné pro PDA, chytré telefony,



Obrázek 12: Nvidia Tegra K1 bude dostupná ve verzi 32bit s 4 jádrovým Cortex A15 procesorem, nebo 64 bitovým 2 jádrovým Denver procesorem [18]

hlasové komprese a dekomprese, 3D grafiku, digitální audio, laserové tiskárny, set-top boxy, automobilové aplikace (například u zabránění zablokování brzd, systém trakce) a další. VFP architektura nenabízí výkon skutečné paralelizace SIMD (jedna instrukce více dat). Proto tato technologie je nahrazována mnohem výkonnějším Advanced SIMD extension - NEON [22][23].

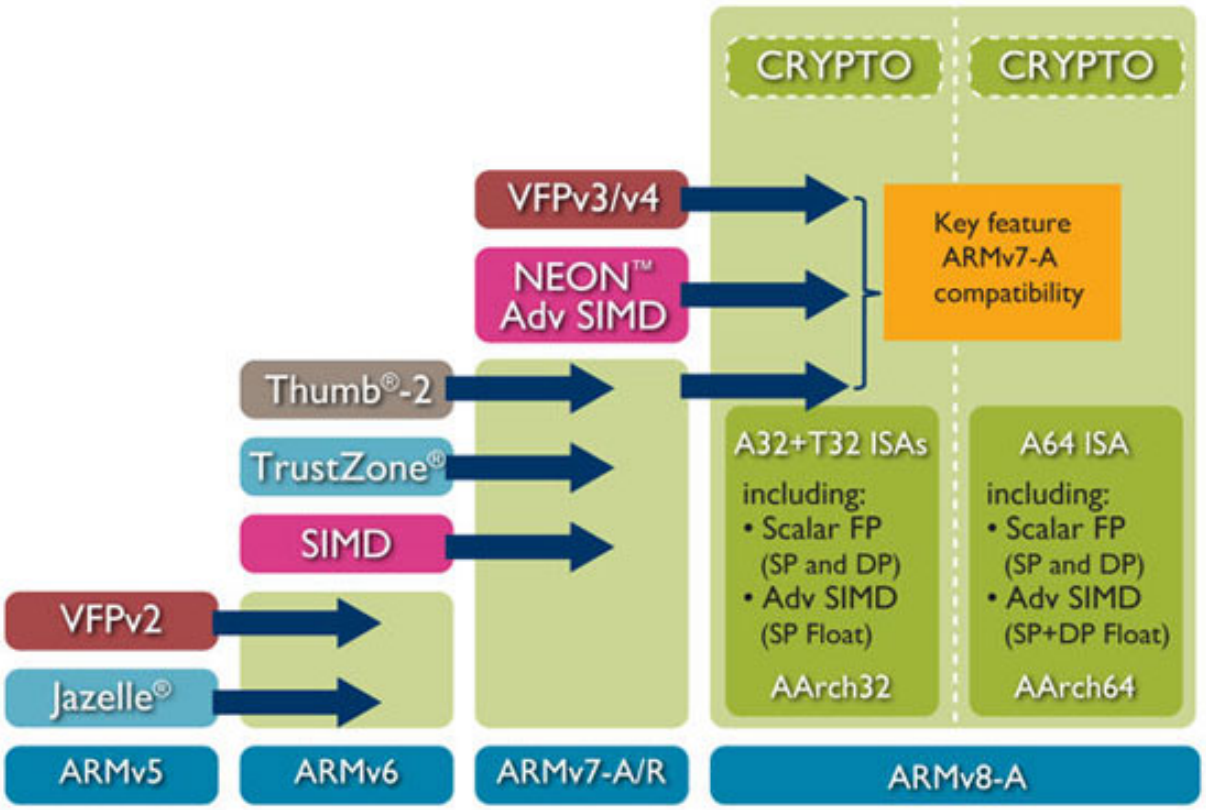
V kombinaci s NEON je technologie VFP používána pro zpracování multimediálních dat, zpracování obrazu, generování písem, 2D a 3D transformace a digitální filtry [23].

Některé procesory (například ty postavené na ARM Cortex-A8) mívají ořezanou verzi, takzvaný VFPLite modul místo plnohodnotného VFP modulu. V tomto případě VFPLite modul pro výpočty s desetinnou řádkou potřebuje více hodinových cyklů [24].

3.3.2.1 VFPv1 Tato verze je již zastaralá. Jde o interní vývojovou verzi [23].

3.3.2.2 VFPv2 Je dobrovolné rozšíření ARM instrukční sady používané u ARMv5TE, ARMv5TEJ a ARMv6 architektury. VFPv2 má 16 64-bitových FPU registrů [23].

3.3.2.3 VFPv3 or VFPv3-D32 Tato verze byla vytvořena pro Cortex-A8 a A9 procesory. Je zpětně kompatibilní s VFPv2, kromě možnosti sledování výjimek při práci s desetinnou čárkou. VFPv3 má standardně 32 64-bitových registrů a přidává VCVT in-



Obrázek 13: Instrukční sady přidávané během vývoje ARM procesorů [21]

strukce pro konverzi mezi datovými typy skalár, float a double. Taky přidává VMOV, která jako konstanta může být načtena do registrů [23].

3.3.2.4 VFPv3-D16 Stejná jako VFPv3 s tím rozdílem, že má pouze 16 64-bitových registrů. Tato verze byla upravena pro procesory Cortex-R4 a R5 (tyto procesory jsou používány jako řadiče disků nebo řídící jednotky automobilů [25])[23].

3.3.2.5 VFPv4 a VFPv4-D32 Vytvořena pro Cortex-A12 a A15 ARMv7 procesory. VFPv4 má standardně 32 64-bitových registrů.

3.3.2.6 VFPv4-D16 Stejná jako VFPv4 s tím rozdílem, že má pouze 16 64-bitový FPU registrů. Implementována na Cortex A5 a A7 procesorech [23].

3.3.3 Advanced SIMD extension - NEON

Technologie NEON je kombinovaná 64 a 128- bitová SIMD instrukční sada, která poskytuje standardizovanou akceleraci pro aplikace zpracovávající signál nebo média. Nabízí komplexní instrukční sadu, oddělení registry a nezávislý prováděcí hardware [26].

NEON podporuje až 16 operací součastně. NEON sdílí stejné registry pro výpočty s desetinnou čárkou jako VFP, ale má nezávislou pipeline [26]. Techlogie NEON je zahrnuta ve všech Cortex-A8 zařízeních a na procesorech Cortex-A9 je volitelné [26].

NEON podporuje SIMD operace pro znaménkové a neznaménkové celá čísla, polynomy s koeficientem o velikosti jednoho bitu a datový typ s plovoucí čárkou o velikosti 32bitů [27].

Existují 2 způsoby jak volat NEON instrukce. Prvním typem volání NEON instrukcí je pomocí C++ rozhraní, kdy instrukce jsou přímo součástí C++ kódu. Druhou možností je volání pomocí assembler instrukcí.

3.3.3.1 Datové typy Neon podporuje 8mi, 16ti, 32 a 64-bitové datové typy pro celá čísla a 32-bitový datový typ pro operace s desetinnou čárkou. Pro polynomiální výpočty má 8mi a 16ti-bitový datový typ poly [27].

Název všech datových typů je tvořen jako <typ><velikost>x<počet_pozic>_t. Jako typ je možné mít pro celočíselné počty se znaménky int a neznaménkové uint. Pro výpočty s desetinnou čárkou slouží float a pro polynomiální výpočty poly. Parament počet pozic udává, kolikrát se daný typ v dané velikosti nachází v registru. Výsledná velikost tohoto datového typu musí vždy souhlasit s tím, jaká je velikost používaného registru [27].

3.3.3.2 S, D a Q registry Základním datovým typem je S registr o velikosti 32-bitů, který je dostupný 32 krát. Tomuto registru se říká jednoslovný a tak vznikl jeho název (single - S). S tímto registrem pracuje pouze pár instrukcí [26].

Velice používaným registrem je D registr. Tento registr má velikost 64-bitů a máme ho 64 krát. Tomuto registru se říká dvouslovný a taky z tohoto názvu má jméno (double - D). S tímto registrem umí počítat většina dostupných instrukcí [26].

Největším registrem je Q registr, který má velikost 128-bitů. D registrů je pouze 16. Říká se mu čtyřslovný a z toho pramení i jeho název (quad - Q). Také s tímto registrem umí počítat většina dostupných instrukcí [26].

3.3.3.3 Instrukce Názvy instrukcí pro C++ rozhraní mají tvar <jméno_operace><flag>_<typ>. Všechny instrukce používají pro typ proměnné zkratku a velikost, aby se vědělo, s jakým datovým typem daná instrukce pracuje. Zkratka pro uint je u, pro int je s, pro float f a pro poly p. Jako flag se používá písmenko q, které nám udává, že budeme pracovat s Q registrem. V případě absence q flagu se používá D registr [28]. NEON neobsahuje instrukce pro dělení.

3.3.3.4 Instrukce pro načtení hodnot do registů Načítání do D registru se provádí pomocí příkazu vld1_<typ><velikost>, vstup této funkce je pole které obsahuje hodnoty,

které se načtou do D registru. Velikost pole musí být minimálně velká, jaký je počet pozic pro danou proměnnou [29]. Pro načtení do Q registru existuje ekvivalentní funkce vld1q_<typ><velikost>[29].

Při výpočtech s NEON je načítání a ukládání hodnot nejdéle trvající instrukcí [29]. Existují i instrukce, které umožňují načtení hodnoty na určitou pozici [29]. A taky instrukce, která nám na všechny pozice načte jednu a tu samou hodnotu.

3.3.3.5 Instrukce pro uložení hodnot z registrů

Uložení hodnot z registrů je podobné jako načítání hodnot do registrů. Pro uložení z D registru se používá příkaz vst1_<typ><velikost>, výstupem je pole obsahující hodnoty vypočítané pomocí NEON [30].

Analogicky pro uložení z Q registru se používá příkaz vst1q_<typ><velikost>[30]. I zde existují instrukce pro načtení hodnoty na určité pozici [30].

3.3.3.6 Instrukce pro sčítání, odčítání a násobení

Instrukce pro sčítání, odčítání nebo násobení mají 2 vstupy, kde vstupem je načtený registr. Velikost vstupních registrů a typ musí být shodný, stejně jako datový typ načtený v registru. Jako výstup dostaneme registr stejného typu, který obsahuje hodnoty, se kterými byla provedena žádaná operace. Název operace pro sčítání je add, pro odečítání sub a pro násobení mul. Operace pracují s D nebo Q registry. U sčítání existují verze, kdy po sečtení dojde k bitovému posunu o 1 doprava. U odečítání existuje verze, jejíž výsledkem je absolutní hodnota rozdílu [31][32][33][34].

Existuje taku speciální verze příkazu, která v jedné instrukci má 3 vstupy a výsledkem instrukce je provedení součtu prvních dvou vstupů a jejich vynásobení třetím. Ekvivalentně existuje operace, která provede odečtení 2 vstupů a vynásobení třetím [34].

3.3.3.7 Instrukce pro bitový posun

Pro bitový posun existuje více typů instrukcí. Základními příkazy jsou instrukce vshr_n_<typ><velikost>a vshl_n_<typ><velikost>pro D registr a vshrq_n_<typ><velikost>a vshlq_n_<typ><velikost>pro Q registr, které provedou posun doleva nebo doprava. Jako první vstup mají registr ve kterém jsou načteny hodnoty, jako druhý vstup mají konstantu o kolik se má provést bitový posun pro všechny pozice stejně.

Další verzí jsou příkazy vshr<flag>_<typ><velikost>a shl<flag>_<typ><velikost>, které mají jako vstup 2 pole stejného typu a bitový posun se provede podle toho, jaký je druhý vstup [35][36].

3.3.3.8 Instrukce pro porovnávání

NEON má tyto instrukce pro porovnávání. Porovnání rovnosti (ceq), větší roven (cge), menší roven (cle), větší (cgt) a menší (clt).

Tyto instrukce mají vždy 2 vstupy. V případě, že pro danou pozici je pravdivé dané porovnávání, tak daná pozice je nastavena jako samé jedničky, v případě nepravdy jako samé nuly.

To znamená, že v případě kdy budeme používat porovnávání pro uint8, tak v případě pravdivosti bude nastavená hodnota 255 a v případě nepravdy 0 [37].

3.4 OS Android

Android je otevřený operační systém určený pro přenosná zařízení (mobily, PDA, tablety) založený na modifikované verzi Linuxového jádra. Jeho zdrojové kódy jsou volně ke stažení a každý si jej může upravit podle sebe a nahrát do jakéhokoliv zařízení, které může následně vyrábět a prodávat bez nutnosti mít licenci [38]. Nainstalovat aplikace jde třemi způsoby a to klasickou instalací přes PC, stažením souboru před internet, nebo pohodlně přes Android market, kde mohou vývojáři zveřejňovat a prodávat své aplikace [39].

3.4.1 Historie

Začátek vývoje systému je datován na říjen roku 2003, kdy byla v Kalifornii založena společnost Android Inc. Tu v roce 2005 koupila společnost Google, kdy se naplno rozjel vývoj tohoto systému [40]. Dlouhou dobu se spekulovalo o tom, že Google vstoupí na trh s mobily, nahlas se o tom začínalo mluvit v září 2007, kdy se zjistilo, že Google podal několik patentů v oblasti mobilních technologií [41].

Obrázek 14: Logo Androidu, oficiální barvou je #A4C639

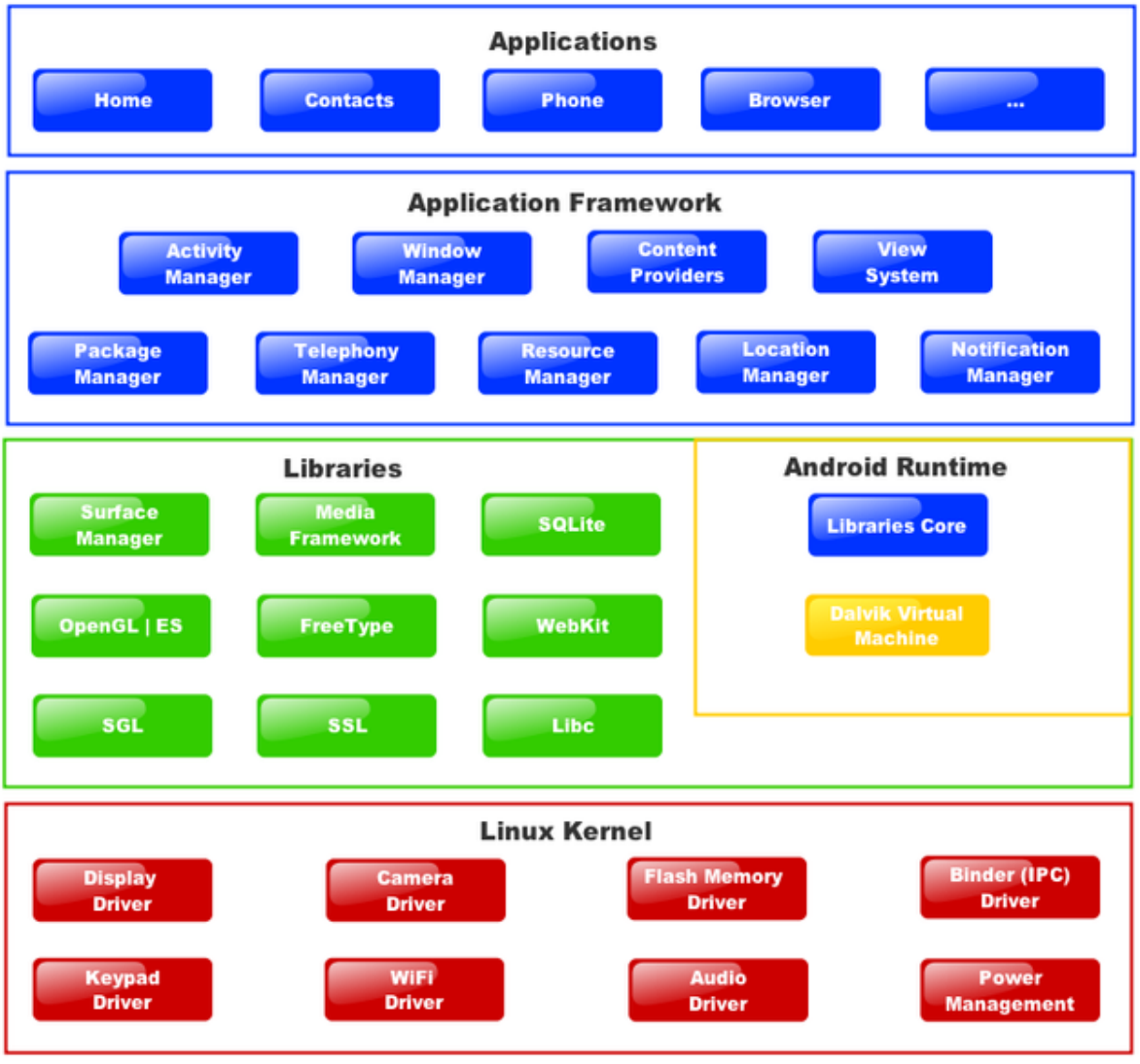
5. listopadu se představilo konsorcium několika firem včetně Google pod jménem Open Handset Aliance [40, 42]. Cílem této aliance je vytvořit otevřené standardy pro mobilní zařízení, ve stejný den představila také tvůj první projekt – Android, mobilní platformu postavenou na verzi jádra Linux 2.6.

23. září 2008 byl oficiálně představen systém Android 1.0 (verze alfa) i s vývojovým prostředím [43, 44, 45], prvním mobilním telefonem prodávaným s OS Android byl T-mobile G1 (HTC Dream) [45]. Od té doby bylo vydáno několik aktualizací, které opravují chyby a přidávají nové funkce. Od verze 1.5 se jednotlivé verze systému se jmenují podle zákusků (Cupcake, Donut, Eclair, Froyo, Gingerbread, Honeycumb – verze pro tablety, Ice cream sandwich, Jelly bean a KitKat).

Při vývoji prošel Android mnoha změn, kdy byl mnohokrát změně vzhled uživatelského prostředí, přidána podpora tabletů (od verze Honeycomb), přidání podpory nativního kódu, USB host, zlepšení odezvy uživatelského prostředí pomocí projektu Butter, podpora více uživatelských účtů, podpora OpenGl a mnoho dalšího.

3.4.2 Architektura

Jak bylo zmíněno, Android je postaven na jádře Linuxu a využívá jeho vlastnosti, umožňuje tak běh více aplikací najednou. Každá aplikace běží jako jeden proces pod vlastním jménem, což vede k izolaci a zvýšení bezpečnosti systému. Uživatelské aplikace běží ve vlastním virtuálním stroji. Pokud jsou na pozadí, operační systém se stará automaticky o to, aby je uzavřel v případě nedostatku systémových zdrojů. Aplikace má několik stavů, ve kterých se může nacházet, než je plně odstraněna z paměti.



Obrázek 15: Schéma architektury systému Android

3.4.3 Vývoj aplikace

Vývoj aplikací se provádí za pomoci Android SDK, které umožňuje vývojářům psát aplikace v jazyce Java s využitím knihoven vyvinutých společností Google. Ale i přesto, že je postaven na jazyce Java, nevyužívá žádného ze standardů jako je Java SE nebo pro mobilní zařízení určený Java ME a taky virtuální stroj pro běh programů napsaných v jazyce Java s názvem Dalvik má odlišnou architekturu od standardního Java virtuálního stroje. Dalvik staví na podmnožině knihoven projektu Apache Harmony, což je open source projekt, který je přepisem technologií Java jako open source. Důsledkem takového řešení je velká nezávislost na původních licencích, které si s sebou původní jazyk i prostředí Java nese. Taky by měl běžet rychleji i na mobilních telefonech s omezeným hardwarem [41, 46].

Oficiálním vývojovým prostředím je Eclipse s nainstalovaným Android Developement Tools Pluginem, který dodá všechny potřebné doplňky pro vývoj aplikací, a taky potřebujeme Android SDK. Pomocí knihoven napsaných Googlem se dostaneme k periferním zařízením, které jsou k dispozici (GPS, kompas, akcelerometr, zapnutí wifi, přístup k SD kartě, atd.). Z bezpečnostních důvodů musí ale taková aplikace zažádat v konfiguračním XML manifestu aplikace o povolení přístupu k periferiím zařízení, instalací programu uživatel tento přístup povolí [47, 48].

3.4.4 Analýza a testování aplikací

Android SDK obsahuje emulátor, ve kterém můžeme testovat naši vytvořenou aplikaci, jeho zapnutí je jednoduché, při tvorbě Android projektu stačí spustit náš projekt, emulátor se spustí sám (pouze při prvním spuštění musíme emulátoru nastavit vlastnosti jaké má emulovat, jako velikost displeje, velikost paměti, atd.), nainstaluje se do něj naše vytvořená aplikace a můžeme ji otestovat. Nevýhodou emulátoru je malá rychlost, proto se v něm nedají testovat real-time aplikace a hry. Taky emulátor neumožňuje testovat aplikace využívající fotoaparát nebo multitouch.

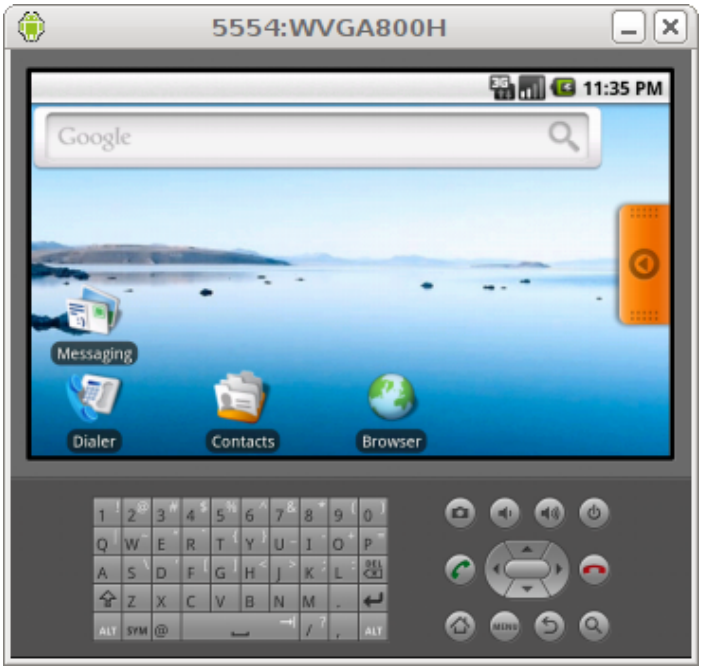
Dalším způsobem je testování na reálném zařízení, k tomu nám stačí zapojit zařízení k PC a povolit v zařízení testování aplikací. Následně pokud spustíme projekt ve vývojovém prostředí, tak dojde k instalaci na zařízení a spuštění aplikace na zařízení. V případě, že používáme vývojové prostředí, které nepodporuje instalaci na zařízení, je možné aplikaci nainstalovat ručním spuštěním adb instalátoru.

Analyzovat naši aplikace můžeme využitím dvou funkcí *Debug.startMethodTracing()* a *Debug.stopMethodTracing()*.

```
Debug.startMethodTracing("cas");
//merena cast kodu
Debug.stopMethodTracing();
```

Výpis 1: Analýza aplikace - měření času

Po spuštění aplikace s daným kódem se na SD kartu uloží soubor cas.trace, který obsahuje informace o volaných metodách a jejich trvání. Pomocí programu *traceview* (který je součástí Android SDK) si soubor můžeme otevřít a ten se zobrazí v grafické podobě. Díky informacím pak můžeme optimalizovat kód aplikace [49].



Obrázek 16: Android emulátor

Pokud máme zařízení připojené během spouštění aplikace k počítači (nebo aplikaci spouštíme na emulátoru), tak v LogCat rozšíření můžeme vidět všechny debug informace.

3.4.5 Google Play

Šíření aplikací probíhá pomocí Google Play (dříve známý pod názvem Android market), což je online obchod aplikací spravovaný Googlem. Aplikace zde umístěné se dají stáhnout a nainstalovat zadarmo, nebo za peníze, vše záleží na vývojáři, který aplikaci do obchodu zařadil. Google si z placených aplikací bere 30% jako poplatek, bohužel pro Českou republiku není zatím prodej aplikací za peníze povolen. Taky je pro vydání aplikací v Android aplikace nutná registrace a zaplacení poplatku $25. Aplikace v marketu se zobrazují jenom uživatelům, jejichž zařízení splňuje podmínky pro instalaci definované v manifestu, takže pokud si definujeme, že zařízení musí mít GPS, tak mobilům bez GPS se aplikace nezobrazí a není je schopen stáhnout [47].

Instalace aplikací přes Google play je nejbezpečnější možností, jak dostat aplikaci do svého zařízení, protože uveřejněné aplikace jsou testovány a i v případě, že se později zjistí, že daná aplikace obsahuje škodlivý kód, tak může být na dálku odstraněna ze zařízení, na kterých byla nainstalována.

3.4.6 2D grafika

Nejčastější API pro 2D grafiku nalezneme v balíčku android.graphics, 2D grafika se kreslí podle toho, čeho chceme dosáhnout. Pokud se jedná o jednoduché vykreslení jednoho obrázku, nebo animace, tak můžeme vykreslit tento obrázek (animaci) do Layoutu, která bude pak vykreslovaná View. Pokud chceme pravidelně překreslovat plochu, tak budeme kreslit do Canvasu, který představuje rozhraní opravdové plochy, na kterou bude kresleno a obsluhuje příslušné Canvas *draw....()* metody (drawPicture, drawText, atd). Pomocí canvasu se kreslí bitmapy na plochu. Abychom mohli vytvořit canvas, tak musíme vytvořit Bitmapu, která bude kreslena. Poté bude canvas kreslit do definované bitmapy. Doporučovaným způsobem je kreslení pomocí funkcí *View.onDraw()* nebo *SurfaceHolder.lockCanvas()* [50].

```
Bitmap b = Bitmap.createBitmap(100, 100, Bitmap.Config.ARGB_8888);
Canvas c = new Canvas(b);
```

Výpis 2: Ukázka nastavení canvasu a bitmapy

Kreslení pomocí View – pokud aplikace nepotřebuje rychlé vykreslování (šachy, pomalu vykreslované aplikace), pak se používá vlastní View komponenta a kreslí se pomocí *Canvasu View.onDraw()*. Pro kreslení stačí pouze dědit z této třídy a implementovat metodu onDraw(), kde se provádí veškerá kreslení na canvas, který Android poskytne o velikosti View. Tato metoda je volána pokaždé, když je požadováno kreslení (Android usoudí, že je čas kreslit, nebo je volána funkce *invalidate()*, která indikuje, že chceme dané View vykreslit, ale nemusí k vykreslení dojít ihned) [50].

Kreslení pomocí SurfaceView – je speciální podtřída View, která poskytuje vyhrazený prostor pro vykreslování s využitím View hierarchie. Vykreslování je obsluhováno v druhém vlákně, takže se nemusí čekat, až je View hierarchie připravena na vykreslení. Pro začátek musíme vytvořit třídu, která rozšiřuje SurfaceView. Třída by měla také implementovat *SurfaceHolder.Callback*. Tato třída je rozhraní, které vám oznámí, pokud dojde k vytvoření, změně nebo zrušení. Tyto události jsou důležité, protože nám indikují, kdy můžeme začít kreslit, nebo skončit s kreslením. SurfaceView objekty se neovládají přímo, ale ovládají se pomocí SurfaceHolder, který se získá pomocí getHolder(). Aby bylo možné z druhého vlákna kreslit, tak musí být canvas zamknut pomocí metody *lockCanvas()*, po vykreslení se uvolní pomocí *unlockCanvasAndPost()*. SurfaceView poté vykreslí canvas, takže je nutné sekvenci zamknutí a odemknutí volat pokaždé, když budeme chtít překreslení. Jedinou výjimkou je, pokud použijeme setType(*SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS*), tehdy nejde používat *lockCanvvas()*, musíme kreslit bez zamčení canvasu, pomocí *postInvalidate()*, pak se chová jako View [50].

3.4.7 Native development kit

Jedná se o způsob, jak v aplikacích spouštět kód vytvořený v C++, takzvané nativní funkce. Tato funkcionalita je nezávislá na výrobci a podporuje načítání dynamických

sdílených knihoven. Hlavním přínosem je zrychlení výpočtu, proto se nativní funkce používají například pro OpenGL, 2D grafiku a matematické funkce.

Pro využití NDK si musíme do počítače NDK zvlášť stáhnout a nainstalovat. Součástí NDK je aplikace Cygwin, která umožňuje spouštět linuxové knihovny na Windows. Pomocí aplikace Cygwin kompilujeme nativní funkce, které se pak stanou součástí naší vytvářené aplikace [51][52]. Aby se zdrojové kódy v C++ správně zkompilovaly, složka jni musí obsahovat správně nadefinovaný soubor Android.mk a Application.mk.

```
//define calling native functions
public native void FaceDetectOpenCV(long matAddrIn);
public native void TemplateMatching(long matAddrIn, long matAddrTemplate);
public native void TemplateMatchingNEON(long matAddrIn, long matAddrTemplate);
```

Výpis 3: Ukázka volání funkce v prostředí Java

Pokud chceme volat nativní funkci z prostředí Javy, tak volání vypadá naprosto stejně, jako bychom volali klasickou funkci, akorát, že nevytváříme tělo funkce a před návratovým typem funkce uvedeme klíčové slovo native. Tím kompilátor bude vědět, že v tomto místě má zavolat nativní funkci. Název nativní funkce v C++ souboru začíná textem Java, následuje názvem našeho balíčku (místo teček jsou podtržítka, názvem naší aktivity a je ukončen názvem funkce, jak ji budeme volat. Vše je spojeno do jednoho velkého názvu, kdy jsou jednotlivé části spojeny podtržítkem.

```
JNIEXPORT void JNICALL Java_cz_vsb_kro224_diplomka_MainActivity_FaceDetectOpenCV(JNIEnv
    *, jobject, jlong);

JNIEXPORT void JNICALL Java_cz_vsb_kro224_diplomka_MainActivity_TemplateMatching(JNIEnv*,
     jobject, jlong, jlong);

JNIEXPORT void JNICALL Java_cz_vsb_kro224_diplomka_MainActivity_TemplateMatchingNEON(
    JNIEnv*, jobject, jlong, jlong);
```

Výpis 4: Ukázka pojmenování funkce v prostředí C++

Funkce napsané v NDK je nejvhodnější volat pro co největší vzorek data (nejlépe předat funkci všechna data, nad kterými má pracovat), protože každé volání nativní funkce znamená přepnutí módu, v jakém je aplikace spuštěna a to se nepříznivě projeví na času zpracování. Nejvhodnějším řešením je tedy předání všech dat nativní funkci, která je zpracuje a zpět vrátí výsledek a nativní funkci voláme znovu až při předání dalších dat. Pokud nad daty chceme volat více funkcí je zase lepší, když funkce se volají uvnitř nativně psaných funkcí, než kdyby data byla vrácena zpět do javy a v ní znovu volána jiná nativní funkce.

3.5 OpenCV

OpenCV je svobodná a otevřená multiplatformní knihovna pro manipulaci s obrazem. Je zaměřena především na počítačové vidění a zpracování obrazu v reálném čase. Původně ji vyvíjela společnost Intel. Knihovnu je možné využít z prostředí jazyků C, C++.

S pomocí různých rozhraní se knihovny OpenCV dají použít i pro jazyky Python, Java, MATLAB/Octave, C#, Ch, Ruby [53].

Od roku 2010 bylo vyvíjeno rozhraní pro požití na GPU podporujících CUDA[54], od roku 2012[55] je vyvíjeno rozhraní pro GPU podporujících OpenCL[56].

Původní knihovna byla vyvíjena pod jazykem C, všechny nové knihovny jsou již vyvíjeny pod C++. Knihovna nyní obsahuje více než 2500 optimalizovaných algoritmů pro počítačové vidění, zpracování obrazu a strojové učení. Tyto algoritmy mohou být použity pro detekci a rozpoznání obličeje, identifikaci objektů, sledování pohybu, vytvoření 3D modelu objektu, spojení více obrázků pro vytvoření obrázky s vysokým rozlišením, nalezení podobného objektu z databáze obrazů, odstranění červených očí při použití blesku, sledování pohybu očí, rozpoznání scény a mnoho dalších [53].

4 Testovací aplikace

V této kapitole popíšu způsob, jak jsem vytvářel testovací aplikaci. Jazyk psaní komentářů ve zdrojovém kódu vytvářené aplikace jsem si vybral angličtinu, i když je tato diplomová práce v češtině. Mezi mé důvody proč jsem se takto rozhodl, patřilo i to, že v případě, že bych tyto zdrojové kódy uveřejnil v budoucnu, tak abych je již nemusel upravovat. Dalším důvodem je i to, že jsem již zvyklý psát komentáře v angličtině z mého zaměstnání a přijde mi to přirozenější.

Aplikace bude fungovat na principu, kdy obraz zachycený kamerou bude zpracován a zobrazen na displeji testovacího zařízení. Funkčnost algoritmů budu ověřovat objektivně podle porovnání výsledků mezi různými verzemi algoritmu i subjektivně dle zobrazeného obrazu (v případě, že algoritmus bude upravovat výstup). Čas výpočtu budu měřit pomocí LogCat funkce ve vývojovém prostředí, zde budu mít i ostatní informace posílané programem.

Základ aplikace bude napsán v jazyku Java, všechny algoritmy pak budou napsány pomocí jazyka C++ v nativních funkcích. Pro hardwarovou akceleraci použiji technologii NEON a instrukce budu volat pomocí interface pro C++.

Pro různé podpůrné funkce je použita knihovna OpenCV, hlavně jsem tuto knihovnu zvolil kvůli třídě Mat, ve které bude uložen jak zdrojový a výsledný obraz tak i různé pomocné obrazy.

4.1 Uživatelské rozhraní aplikace a ovládání

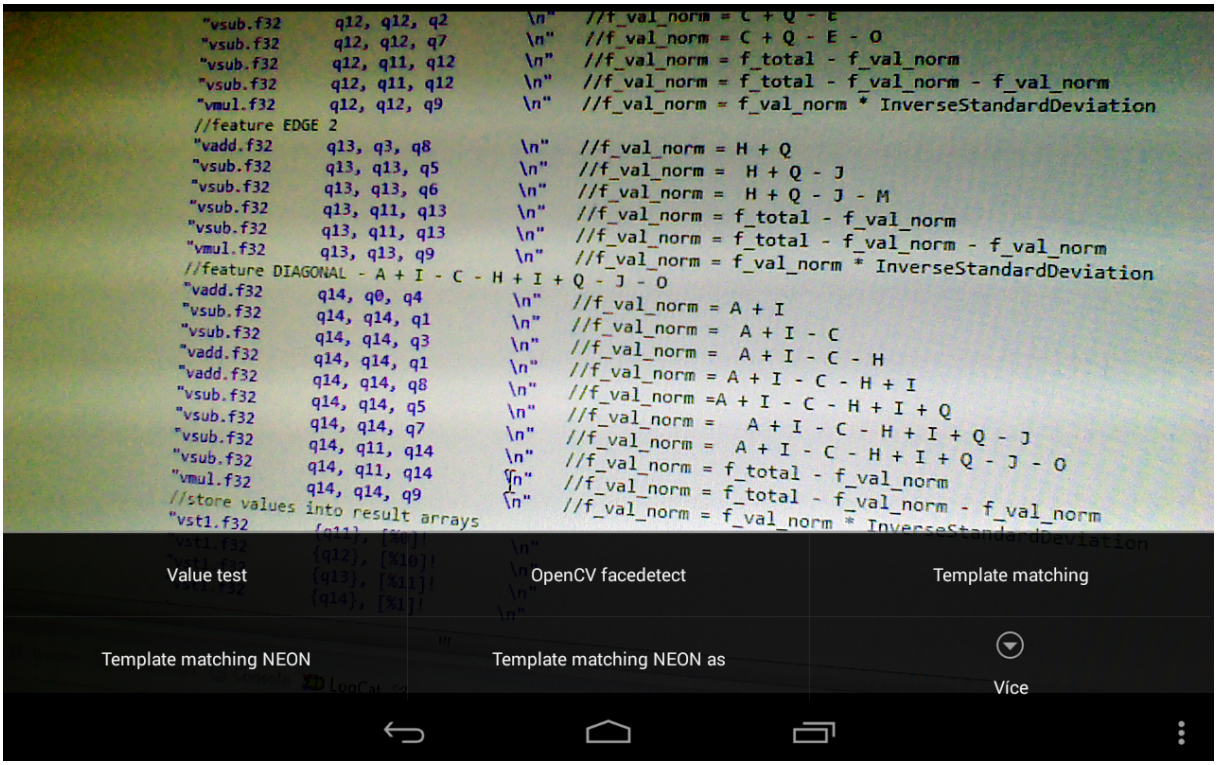
Po spuštění aplikace se zobrazí obraz zachycený kamerkou (přední v případě, že zařízení má jenom přední kamerku, zadní v případě, že zařízení má zadní kameru). Úvodní algoritmus, který se zobrazí jako první je vyhledání obličeje v obrazu pomocí knihovny OpenCV. Jde o ukázku toho, jak by výstupní obraz (v případě, že jejich cílem je vyhledání obličeje) měl přibližně vypadat.

Výběr druhu algoritmu probíhá pomocí tlačítka Možnosti, které se nachází v pravém dolním rohu a má tvar tří teček nad sebou. Po zmáčknutí na tlačítko možnosti se objeví panel, na kterém se nachází tlačítka s možnostmi výběru algoritmu, který chceme spustit. V případě, že je možností více, než je možno zobrazit na displeji zařízeni (aplikace ukazují maximálně 2 řádky možností), tak se jako poslední objeví tlačítko více a po jeho zmáčknutí se nám objeví dodatečné menu se zbývajícími možnostmi, které je skrolovací. Ukázku uživatelského rozhraní vidíme na obrázku 17.

Po vybrání možnosti v menu se zapne zvolený algoritmus a dojde ke zpracování vstupního obrazu daným algoritmem.

4.2 Získání obrazu z fotoaparátu

Obraz fotoaparátu je získán pomocí implementace interface CvCameraViewListener2, který je součástí knihovny OpenCV. Zvolil jsem tuto možnost z toho důvodu, že na rozdíl od SurfaceView, které je přímo implementováno jako součást knihoven Android pomocí



Obrázek 17: Ukázka uživatelského rozhraní pro výběr, jaký algoritmus bude použit pro zpracování vstupního obrazu.

CvCameraViewListener2 ve funkci onCameraFrame získávám obraz již jako objekt CvCameraViewFrame, který jde převést na třídu Mat. Následně si zjistím adresu v paměti pro danou instanci objektu Mat a adresu pošlu do nativní funkce.

V nativní funkci si pak z adresy objektu získám odkaz na třídu Mat a můžu vstupní obraz zpracovávat. Stejným způsobem získám odkaz na další vstupní nebo výstupní obrazy, popřípadě pole.

4.3 Registrace nativní knihovny

Aby aplikace mohla spouštět nativní funkce, tak musí být v javovském kódu zaregistrovány používané nativní knihovny obsahující dané nativní funkce. Na ukázce kódu číslo 5 je uveden příkaz kterým se zaregistruje knihovna. V testovací aplikaci je knihovna načtena až v případě, kdy je úspěšně načtená knihovna OpenCV a vytvořeno SurfaceView (tedy aplikace je připravena přijímat obraz). Nejvhodnější je načítat knihovnu ve funkci onManagerConnected, tehdy je jisté, že byla úspěšně načtena knihovna OpenCV a úspěšně vytvořeno spojení s fotoaparátem zařízení.

System.loadLibrary(”kro224_diploma”);

Výpis 5: Registrace knihovny s nativními funkcemi

K zaregistrování knihovny s nativními funkcemi musí být v první řadě vytvořena knihovna daného jména a taky se musí v Android.mk souboru nastavit parametry s jakými budou zdrojové kódy kompilovány a jaké knihovny k tomu budou použity. V ukázce kódu číslo 6 lze vidět kompletní ukázku, jak nastavit Android.mk soubor, podle kterého budou C++ zdrojové kódy kompilovány. V nastavení jsou vybrány všechny soubory, které budou zkompilovány a knihovny, které budou ke kompilaci použity.

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)
// jmeno knihovny
LOCAL_MODULE := kro224_diploma
// soubory v knihovne
LOCAL_SRC_FILES :=jni_part.cpp
LOCAL_SRC_FILES += face_detect.cpp

// pouzite knihovny ke kompilaci
LOCAL_LDLIBS += -llog -ldl

include $(BUILD_SHARED_LIBRARY)
```

Výpis 6: Obsah souboru Android.mk

Proto, aby bylo možné logovat a posílat vlastní debug informace přímo z nativního kódu musí být použita knihovnu log. Na výpisu kódu číslo 7 lze vidět knihovnu, kterou musíme naimportovat, definici příkazu a ukázku volání vytvořené funkce. Funkce pro logování se musí ale volat opatrně, protože volání této funkce zpomalí běh celého programu a výrazně to zkresluje výsledky. Tyto funkce tedy využíváme na logování speciálních a chybových stavů. Tyto logovací informace pak čteme pomocí pluginu LogCat ve vývojovém prostředí.

```
#include <android/log.h>      // import knihovny

// definice prikazu
#define LOGI(...)  __android_log_print (ANDROID_LOG_INFO,"kro224-diplomka",__VA_ARGS__)

LOGI("Zprava");
```

Výpis 7: Logování v nativních funkcích

4.4 Použití NEON instrukcí v nativních funkcích

Android NDK od verze 3 podporuje ABI verzi armeabi-v7a, která sebou přinesla i podporu Thumb-2 a VPFv3 na všech zařízeních a podporu dobrovolného rozšíření ARM instrukcí přezdívané NEON [57].

K tomu, aby bylo možné použít NEON instrukce v Android aplikaci, musí být splněno několik dále uvedených podmínek. V souboru Application.mk musí být nastaven ABI mód (viz ukázka kódu číslo 8).

```
APP_ABI := armeabi-v7a
```

Výpis 8: Definice ABI verze

Taky je vhodné naimportovat do c souborů knihovnu arm_neon.h, která obsahuje funkce, pomocí kterých zjistíme, jestli zařízení je schopné spouštět NEON instrukce. Tím je možné zajistit, že v případě podpory NEON instrukcí se na zařízení spustí kód hardwarově akcelerovaný pomocí NEON instrukcí, v jiném případě se spustí standardní C++ kód [57]. Ukázku takového rozhodování vidíme na ukázce kódu číslo 9.

```
if (android_getCpuFamily() == ANDROID_CPU_FAMILY_ARM &&
    (android_getCpuFeatures() & ANDROID_CPU_ARM_FEATURE_NEON) != 0)
{
    // pouziti NEON instrukci v pripade podpory NEON
    ...
}
else
{
    // pouzici C++ verze v pripade ze NEON neni podporovan
    ...
}
```

Výpis 9: Rozhodování jaký algoritmus použít

V souboru Android.mk musí být nastaveno, že daná aplikace podporuje neon a musí být naimportována knihovna cpufeatures. Potřebné příkazy lze vidět na výpisu kódu číslo 10. Nyní je připraven základ testovací aplikace [57].

```
LOCAL_STATIC_LIBRARIES := cpufeatures

LOCAL_ARM_NEON := true

$(call import-module,cpufeatures)
```

Výpis 10: Potřebné příkazy pro správnou kompilaci zdrojových kódů s podporou NEON instrukcí

4.5 Použití NEON assembler instrukcí v nativních funkcích

K tomu, aby bylo možné použít NEON assembler instrukce, musí být nějakým způsobem vložen kód napsaný v assembleru do našeho C++ zdrojového. Tomuto stylu volání assembler instrukcí se říká vkládaný assembler (anglicky inline assembler). Vkládaný assembler nám umožňuje vložit assembler příkazy mezi příkazy vyššího programovacího jazyka. To se využívá hlavně při používání SIMD instrukcí, nebo když chceme získat přístup k instrukcím procesoru (V/V porty, systémové volání), které překladač vyššího programovacího jazyka neumožňuje volat. Volání vkládaného assembleru se provádí pomocí příkazu asm, který provede vykonání našeho assembler kódu V případech, kdy

klíčové slovo asm může dělat problémy [58] se používá příkaz __asm__. Příklad je uveden na ukázce kódu 11. Kompilátoru se již nemusí nastavovat žádné další parametry, vše potřebné je součástí knihovny arm_neon.

Jako vstup __asm__ funkce je vložen NEON assembler zdrojový kód, následné další parametry se nacházejí za dvojtečkami. Za první dvojtečkou jsou nastaveny odkazy na pouze výstupní proměnné, za druhou dvojtečkou jsou odkazy všechny vstupní nebo výstupní proměnné, za třetí dvojtečkou se nachází používané registry ve vloženém assembler kódu [58][59].

Na vstupy a výstupy se odkazuje pomocí % odkazu, kdy číslo za % nám říká o kolikátý vstup/výstup se jedná. V assembleru se nepoužívají vlastní proměnné, ale přímo použité registry, když tedy chceme použít D registr, tak požijeme proměnnou d a číslo registru, který chceme použít a naprosto stejně použijeme Q registr s písmenkem q. Důležité je, aby registr byl použitelný (zaregistrovaný) a aby daný registr existoval [59].

V případě chyby je ale kompilátor schopný objevit velké množství standardních chyb. Neumí ale upozornit v případě, když se spleteme a zvolíme jiný datový typ.

```
__asm__ volatile(
        "vld1.u8    {d1}, [%1]      \n"
        "vld1.u8    {d2}, [%2]      \n"
        "vld1.u8    {d5}, [%3]      \n"
        "vcge.u8    d3, d2, d1      \n"   //d3 = d1 >= d2
        "vshr.u8    d3, d3, #7      \n"   //d3 = d3 >> 7
        "vmul.u8    d4, d3, d5      \n"   //d4 = d3*d5
        "vst1.u8    {d4}, [%0]!     \n"
        :
        :"r"(d),
         "r"(gc_),
         "r"(gp),
         "r"(a)
        :"d1", "d2", "d3", "d4", "d5"
         );
```

Výpis 11: Použití NEON assembler instrukcí v C++ zdrojovém kódu

4.6 Kontrola správnosti výpočtů

Pro kontrolu správnosti dat si v menu zvolíme možnost Value Test. V tomto případě se postupně spouští všechny algoritmy a provádí se kontrola, jestli vypočítaná data pro různé algoritmy jsou shodná. Výsledek kontroly dat se po skončení kontroly vypíše na displeji zařízení.

5 Implementace algoritmů

V této kapitole se budu věnovat implementaci jednotlivých vybraných algoritmů ze zpracování obrazu a jejích následnou paralelizací pomocí technologie Advanced extension SIMD - NEON. Implementace bude provedena na testovací aplikaci v programovacím jazyku C++, tedy nativně.

Jako vybrané algoritmy, které budu implementovat, jsem si vybral Porovnání vzorů, kdy se budu snažit hardwarově akcelerovat stěžejní část algoritmu a to výpočet integrálního obrazu. Dalším algoritmem budou lokální binární vzory, kde se budu snažit porovnat rozdíl při využití D a Q registrů a zjistit kolik času ušetříme, pokud se budu snažit minimalizovat počet dat načítaných z paměti RAM do registrů NEON. Posledním vybraným algoritmem je výpočet Haarových příznaků.

5.1 Integrální obraz

5.1.1 Implementace v C++

Pro výpočet sumy pro pixel, kterou vypočítáme, použijeme všechny hodnoty pixelů, které se nachází vlevo a nahoru od aktuálního pixelu. Teď je velice dobré si uvědomit, že se jedná o hodnoty, které již byly počítány. Při implementaci hodnotu aktuálního pixelu počítám pomocí vzorce:

$$I(x,y) = i(x,y) + I(x-1,y) + I(x,y-1) - I(x-1,y-1) \tag{12}$$

Tím zajistím, že výpočet hodnoty aktuálního pixelu vypočítám sečtením tří hodnot a odečtením jedné. Toto řešení je určitě mnohem výhodnější, než procházet všechny pixely vlevo a výše a dělat jejich součet.

```
for radky, sloupce {
        if (y == 0 || x == 0)
        {
            IntegralniObraz(x,y)= 0;
        }
        else
        {
            IntegralniObraz(x,y)= ZdrojovyObraz(x−1,y−1) + IntegralniObraz(x−1,y) +IntegralniObraz
                (x,y−1) − IntegralniObraz(x−1,y−1);
        }
}
```

Výpis 12: Zjednodušený algoritmus použitý pro výpočet Integrálního obrazu

Zdrojový i integrální obraz máme uložený ve formátu Mat (třída knihovny OpenCV), kde se k bodům přistupuje pomocí funkce at, která přijímá argumenty x a y. Pro průchod přes všechny pixely použijeme dvě vnořené smyčky for. V každém cyklu provedeme v případě, že osa x nebo y není 0 součet dle vzorce uvedeného výše.

Jako datový typ používám double a long. Zajímá mě rozdíl výkonu při počítání u těchto 2 datových typů. Oba jsou 64-bitové a oba jsou tedy schopné bez problému pojmout

sumu obrazu o velikosti 1280x720 (generovaný kamerou testovaného zařízení) i kdyby všechny pixely byly nastaveny na hodnotu 255. Porovnání rychlosti těchto 2 algoritmů je uvedeno v kapitole 5.1.4.

5.1.2 Hardwarová akcelerace pomocí NEON C++

Při psaní tohoto hardwarově akcelerovaného algoritmu jsem narazil na jeden zásadní problém. Zatímco pro C++ jsem mohl použít 64-bitový datový typ pro výpočty s plovoucí desetinnou čárkou, tak NEON podporuje maximálně 32-bitový datový typ float32_t. Pro celá čísla NEON podporuje jak datový typ int32_t, tak i int64_t. Z toho důvodu jsem se rozhodl, že udělám tři verze pro NEON a v každé z nich si zkusím použít jiný datový typ.

Při implementaci tohoto algoritmu jsem narazil na jednu nepříjemnost, která ve většině případů vylučuje možnost použití SIMD výpočtů. Hodnota aktuálně počítaného pixelu je totiž závislá na hodnotě předchozího pixelu.

V daném případě je hodnota počítaného pixelu závislá na 3 pixelech a své vlastní hodnotě ze zdrojového obrazu, takže SIMD instrukce použijeme na sečtení a odečtení u 2 hodnot z integrálního obrazu, které se nachází o řádek výše a jsou tedy již vypočítány a hodnoty zdrojového obrazu. Poslední součet s předchozí hodnotou provedu na konci již pomocí C++ bez využití NEON.

5.1.2.1 Datový typ float32_t

Na začátek jsem si spočítal, kolik registrů budu potřebovat pro výpočet integrálního obrazu.

Potřebuji tedy 1 registr pro načtení hodnoty z originálního obrazu, 2 registry pro načtení hodnoty z již vypočítaného registru. Další jeden registr, který do kterého budu ukládat výsledek a budu ho používat na součty a odečítání. Celkově tedy budu potřebovat 4 registry.

Pro daný algoritmus jsem se rozhodl použít Q registr, který mám dostupný 16 krát. V případě, že na výpočet potřebuji 4 registry, tak jej můžu použít 4 krát. V každém Q registru o velikosti 128-bitů budu mít 4 krát 32-bitový datový typ float. Tímpádem během jednoho cyklu vypočítám 16 hodnot.

Taky musím ošetřit situaci, kdy počet sloupců nebude dělitelný číslem 16. Pokud nastane tokový případ, pak hodnoty zbývajících pixelů budou spočítány pomocí čistého C++ algoritmu.

5.1.2.2 Datový typ int32_t

Jde o shodný algoritmus jako v případě předchozí kapitoly s tím rozdílem, že místo datového typu float32_t použiji celočíselný datový typ int32_t. Při použití očekávám zrychlení jenom díky domu, že se jedná o celočíselný datový typ.

Na našem testovacím zařízení je použití datového typu int32_t možné z toho důvodu, že zařízení předává obraz do naší aplikace v rozlišení 1280x720 (rozdíl oproti 1280x1024, které je schopna kamera snímat). Tedy když obrázek zpracovávám ve stupních šedi, tak i kdyby měly všechny pixely maximální hodnotu 255, tak maximální možná suma by byla

235 008 000, což je číslo stále menší, než pojme 32-bitový celočíselný datový typ, jehož maximální možná hodnota je 2 147 483 647 (v jeho znaménkové formě).

5.1.2.3 Datový typ int64_t Jedná se o verzi, která bude vhodná pro velké obrázky, ale nevýhodou této verze bude její rychlost. Vzhledem k tomu, že Q registr má 128 bitů, tak do jednoho Q registru můžu načíst pouhé 2 proměnné. Proto v jednom cykly vypočítám pouhých 8 hodnot oproti 16ti u float32_t a int32_t.

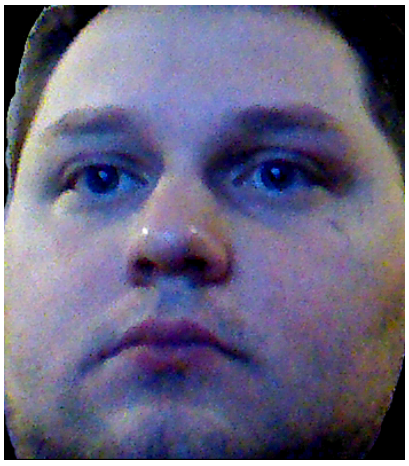
Jedná se tedy o úpravu algoritmu použitý pro datový typ int32 použitý v kapitole 5.1.2.2. Očekávaný výsledek je ten, že použití datového typu int64_t bude pomalejší než u int32_t, ale pro větší obrázky nedojde k přetečení.

5.1.3 Hardwarová akcelerace pomocí NEON assembleru

Pro použití NEON assembler instrukcí jsem si vybral datový typ int32_t, jehož hardwarově akcelerovaná verze pro výpočet integrálního obrazu se počítá nejrychleji (více informací v následující kapitole 5.1.4). Funkčně se jedná o shodný algoritmus jako v případě kapitoly 5.1.2.2 a přepis z NEON C++ instrukcí na NEON assembler instrukce s tím rozdílem, že z důvodu bugu v gcc kompilátoru [60] použitého v Cygwin jsem mohl počítat pouze 12 hodnot v jednom cyklu. V daném případě tedy použiji jenom 12 Q registrů.

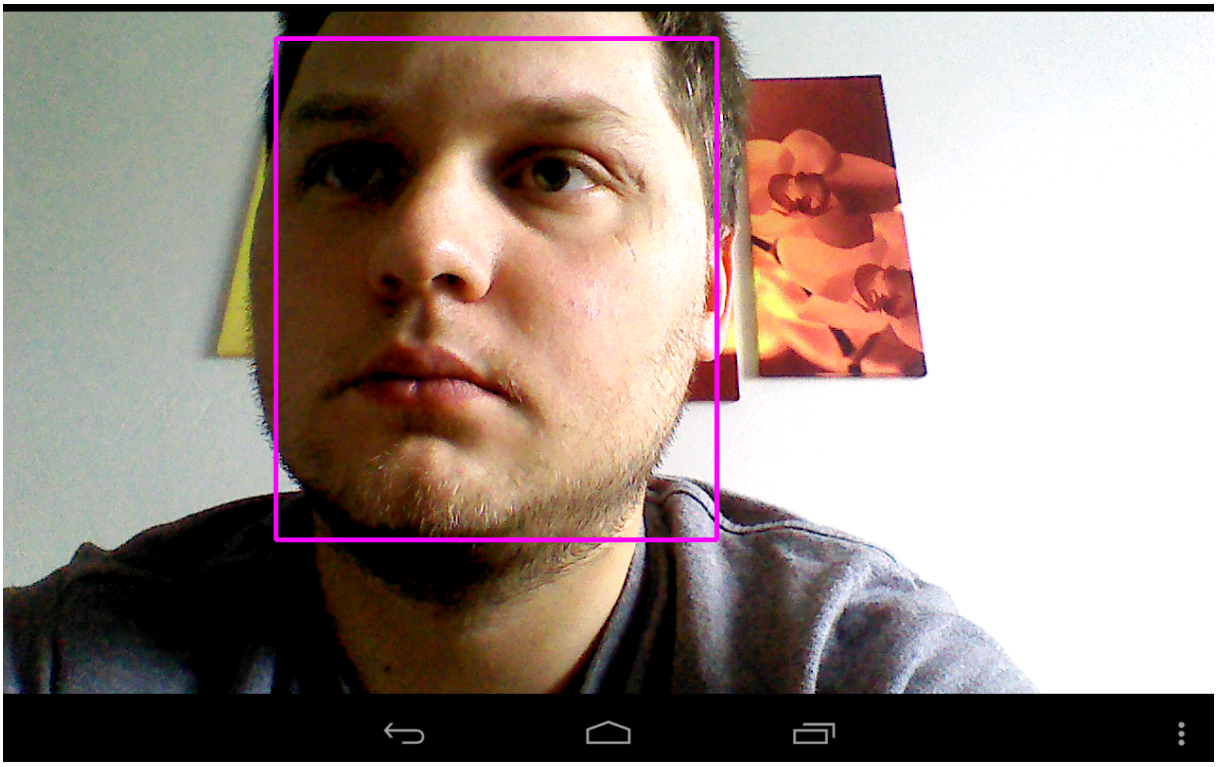
5.1.4 Porovnání rychlosti algoritmů pro výpočet Integrálního obrazu

Na testovacím zařízení Nexus 7 jsem pro implementované algoritmy naměřil časy uvedené v tabulce 3. Pro měření jsem ignoroval prvních 100 cyklů a spočítal jsem až čas potřebný k provedení dalších 500 zpracování algoritmu. Tím jsem si zajistil, že časy prvních výpočtů nejsou brány v úvahu (zapínání jader, atd.) a nemůžou mi tedy ovlivnit výsledek.



Obrázek 18: Použitý vzor pro algoritmus hledání vzorů

Jako vzor, který bude použit pro hledání na obrazu získaného z kamery zařízení, jsem použil výřez mého obličeje, který je ukázaný na obrázku 18. Místa, kde zpoza obličeje bylo vidět pozadí, jsem nahradil černou barvou. Výsledek porovnání vzorů a vyhledání místa s nejmenším SAD na obraze je viditelný na obrázku 19.



Obrázek 19: Vyhledání tváře podle vzoru

Když se podívám na tabulku 3 s dosaženými časy, tak zjistíme, že nejrychlejší byl algoritmus napsaný v C++ s datovým typem long. Byl rychlejší skoro o 3%, než verze s datovým typem double.

	C++ double	C++ long	NEON float32_t	NEON int32_t	NEON int64_t
Celkem [ms]	99482,2	96692,2	146949	138461	160258
Průměr [ms]	198,96	193,38	293,9	276,92	320,516
Zrychlení [%]	-2.89	0	-51,98	-43,20	-65,74

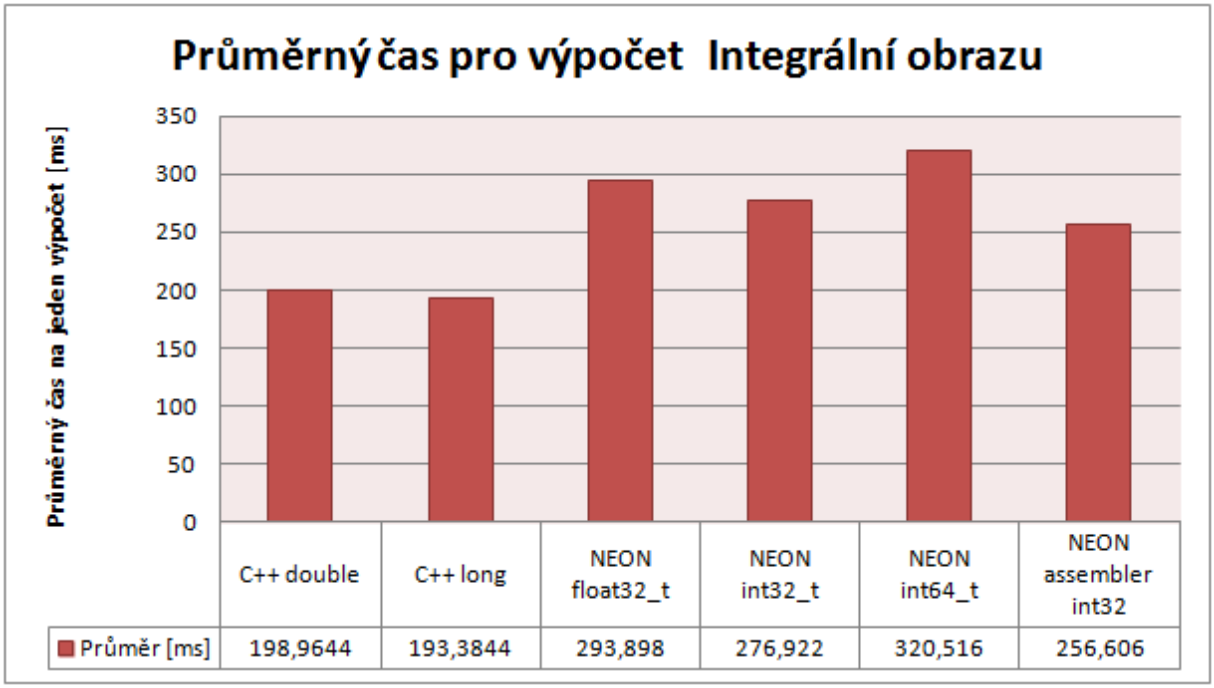
Tabulka 2: Porovnání časů pro zpracování 500 snímků

Zato verze hardwarově akcelerované pomocí NEON byly pomalejší. Nejrychlejší hardwarově akcelerovanou verzí pomocí NEON C++ instrukcí byla verze s datovým typem int32_t, která byla o více než 43% pomalejší, než verze s datovým typem long.

	NEON assembler int32_t
Celkem [ms]	128303
Průměr [ms]	256,606
Zrychlení [%]	-32,69

Tabulka 3: Porovnání časů pro zpracování 500 snímků při použití NEON assembler instrukcí

Rychlost výpočtu hardwarově akcelerovaného algoritmu jsem se snažil ještě urychlit využitím NEON assembler instrukcí. Z předchozích výsledků jsem věděl, že nejrychlejší implementace je pomocí int32_t, proto jsem assembler verzi vytvořil již jenom pro tento datový typ. Verze napsaná pomocí NEON assembler instrukcí se stala nejrychlejší hardwarově akcelerovanou verzí pro výpočet Integrálního obrazu, která ale pořád byla pomalejší, než čistá C++ verze a to o více než 32%. Ale použitím NEON assembler instrukcí jsem snížil čas oproti NEON C++ instrukcí o skoro 8%. Na tomto případu jde vidět, že algoritmy, kde je výpočet závislý na předchozím výsledku je neefektivní pokoušet se provádět pomocí SIMD výpočtů. U toho algoritmu jsem přípravou vstupních polí pro načtení do registru strávil většinu času, který trvá výpočet pomocí C++.



Obrázek 20: Graf průměrných časů pro výpočet Integrálního obrazu

Nejpomalejším výpočtem byl výpočet hardwarově akcelerován pomocí NEON s datovým typem int64_t. Vzhledem k tomu, že se do jednoho Q registru vešly jenom 2 hodnoty,

tak nedocházelo k velké paralelizaci u výpočtu. Příprava pole pro vstup dat byla hlavně v tomto případě o hodně více náročná, než kolik jsem následně ušetřil pomocí paralelního výpočtu.

Pro to, abych mohl dosáhnout lepších výsledků, bych potřeboval více Q registrů, abych mohl provádět více hardwarově akcelerovaných operací, než začneme provádět poslední sečtení pomocí C++. Popřípadě kdyby existovaly registry, které by měly větší velikost.

Kontrolu funkčnosti výpočtu integrální obrazu jsem ověřil na náhodně generovaném obraze. Pro ověření jsem si zavedl kontrolu hodnoty posledního vypočítaného bodu. U všech algoritmů jsem došel ke stejnému výsledku pro shodný vstup.

Subjektivní ohodnocení funkčnosti porovnání vzorů je takové, že vyhledání funguje nejlépe na jednolitém pozadí světlé barvy. Stávalo se mi, že v případě že jsem byl velice nasvícený a na kameru ukázat nějaký tmavý předmět, tak algoritmus ohodnotil pozici, kde se nacházel tento tmavý předmět jako místo s nejmenším SAD.

5.2 Lokální binární vzory

5.2.1 Implementace v C++

Implementace algoritmu v C++ vypadá následovně. Jako první krok si musím načíst obrázek ve stupních šedi. V testovací aplikaci je to upraveno tak, že do nativní funkce pro výpočet LBP se již posílá obrázek zachycený kamerou ve stupních šedi.

V nativní funkci obraz procházím pixel po pixelu s tím, že vynechávám okolní body. Vzhledem k tomu, že obraz mám uložený ve formátu Mat (třída knihovny OpenCV), kde se k bodům přistupuje pomocí funkce at, která přijímá argumenty x a y, tak použiji dvě vnořené smyčky for na procházení obrazu. V rámci zjednodušení (při následné paralelizaci) jsem si zavedl podmínku, že u LBP bude vždy parametr $P = 8$. Tímto jsem si taky zajistil, že část funkce 2^p, bude v našem případě maximálně $2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$, tedy maximální dosažitelná suma bude 255. Při použití datového typu uchar (neznaménkový, 8-bitů, maximální hodnota 255) nemusím následně provádět normalizaci.

Obraz získaný pomocí LBP je následně zobrazen na displeji zařízení.

```
//vsechny radky bez krajnich radku
for (int y = R; y < radky_obrazu − R; y++) {
    //vsechny sloupce bez krajnich sloupcu
    for (int x = R; x < sloupce_obrazu − R; x++) {
        //vsechny okolni body kolem gc
        for(int p = 0; p < 8; p++)
            // ohonoceni gc, ktere bude ulozeno do vystupniho obrazku, ziskano jako suma
                ohodnoceni okolnich bodu
            sum = (gc >= gp) * mocnina(2,p);
    }
}
```

Výpis 13: Zjednodušený algoritmus použitý pro výpočet LBP

Algoritmem uvedeným v ukázce kódu procházím všechny pixely. Následně porovnávám okolní body g_p ve vzdálenosti R s bodem g_c. V případě, že g_c je větší, než g_p, tak získám mocninu čísla 2 pro danou pozici g_p vůči g_c (ukázka vah ohodnocení pixelů je v tabulce 1). Sečtu všechny získané ohodnocení a tím získám LBP ohodnocení pro g_c. Toto ohodnocení uložíme do výstupního obrazu. Jako poslední úpravu jsem provedl nahrazení násobení za bitový posun.

5.2.2 Hardwarová akcelerace pomocí NEON C++

V této kapitole se budu věnovat způsobu, jakým jsem paralelizoval algoritmus LBP. Vyvinul jsem 4 verze. Ve všech čtyřech verzích používám pro ukládání 8mi bitový datový typ uint8_t. Pro načtení řádků slouží jeho řádková verze, kdy pro práci s 64 bitovým D registrem slouží datový typ uint8x8_t, pro práci s 128-bitovým Q registrem slouží datový typ uint8x16_t.

Nevýhodou NEON je omezený počet používaných instrukcí, tedy to, že paralelizovat jde jenom určité části algoritmu a taky jenom to co se vleze do jednoho D nebo Q registru.

5.2.2.1 Verze 1 Při prvním využití paralelizace pomocí NEON jsem vytvořil algoritmus podobný originálnímu algoritmu pro LBP.

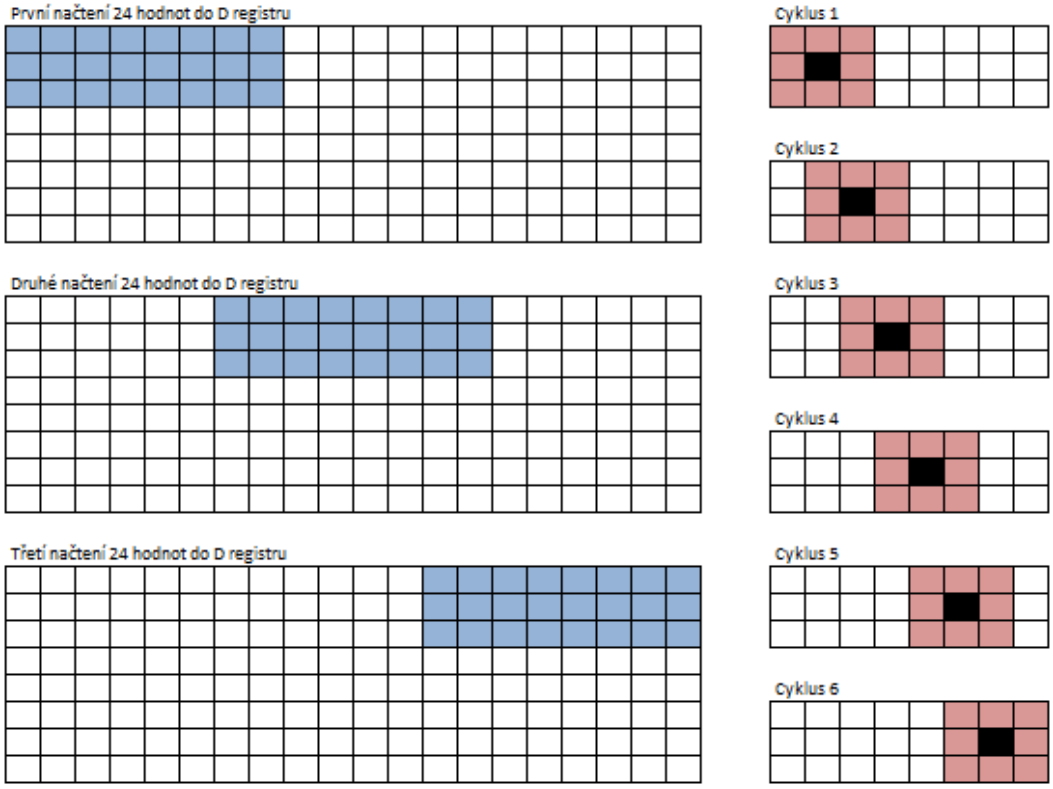
V tomto algoritmu si načtu do zásobníku D typu 8 okolních bodů. Abych mohl těchto 8 bodů porovnat najednou a využít jedné instrukce na více dat, tak si musím načíst do D registru 8 krát hodnotu centrálního bodu, to se nejlépe provede pomocí příkazu vdup_n_u8, který jako jediný parametr očekává hodnotu, kterou nastaví na všechny možné pozice. Pomocí operace vcge_u8 dojde k porovnání, jestli je větší g_c nebo g_p. V případě, že g_c je menší, tak výstupní D registr bude na daném místě obsahovat číslo 0, v případě, že g_c bude větší, tak se uloží samé jedničky, tedy u 8mi bitového datového typu to bude číslo 255.

Protože kombinace 0 a 255 není správná (a NEON mezi instrukcemi nemá dělení, které by se dalo využít pro získání korektní váhu), tak musím v prvním kroku upravit tyto dvě hodnoty na 0 nebo 1. Pro získání 0 a 1 jako výstup porovnání, provedu pomocí funkce vshr_n_u8 bitový posun o 7 doprava, kdy při bitovém posunu čísla 0 zůstane pořád číslo 0 a při bitovém posunu o 7 doprava u čísla 255 dojde ke změně na číslo 1.

Výsledné číslo pomocí funkce vmul_u8 vynásobím váhou, kterou jsem si načetl na začátku do D registru.

5.2.2.2 Verze 2 Vzhledem k tomu, že při předchozí paralelizaci dochází k tomu, že pokud máme R=1, tak 6 z 9 pixelů bude při následujícím výpočtu znovu načítáno. Proto jsem vytvořil algoritmus, kdy dojde ke skupinovému načtení dat. Taky jsem testováním přišel na to, že načítání dat do registru je velice pomalé a proto se snažím minimalizovat načítání hodnot do D nebo Q registru.

V tomto kroku pominu proměnnou R, protože daný algoritmus bude upravený pro $R = 1$. Pokud máme $R = 1$, tak vím, že vždy budu potřebovat právě 3 po sobě jdoucí řádky. Data budu načítat do 3 64-bitových D registrů po 8mi pixelech (datový typ



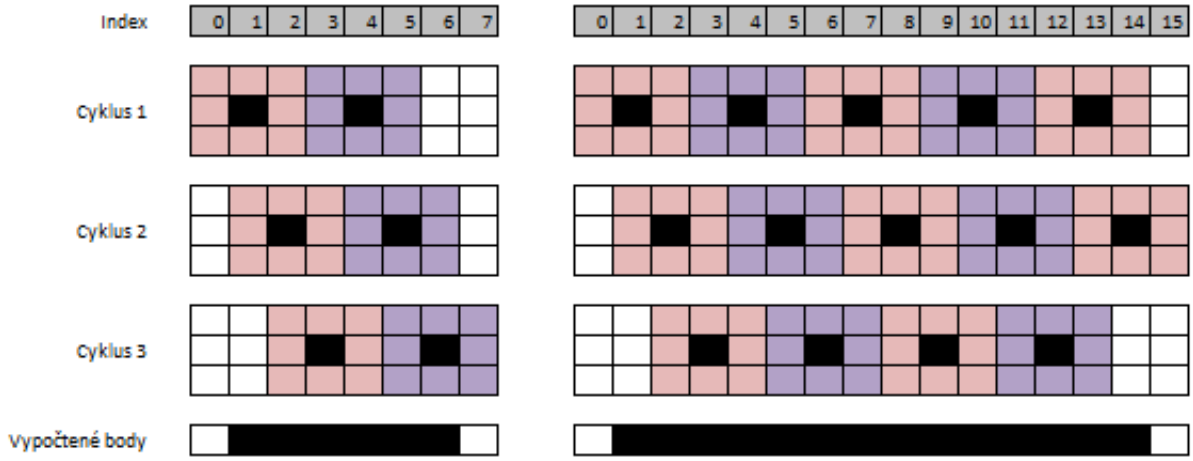
Obrázek 21: Ukázka průběhu výpočtu paralelního algoritmu pomocí NEON u verze 2. Body načítáme jako 3 řádky po 8mi bodech, celkem 24 bodů a z těchto načtených bodů vypočteme 6 hodnot LBP.

uint8x8_t), v zásobníku tedy budu mít načtených 24 pixelů a použiji je pro výpočet 6ti bodů najednou (nepočítá se vždy první a poslední bod z důvodů chybějící informace). Při načítání dalších 24 bodů tedy dojde k tomu, že je načteno znovu jenom 6 bodů z 24 (1/4 oproti 2/3).

Ve chvíli, kdy jsou data načtena, tak do D registru načtu 8x g_c bod a všechny body na všech 3 řádcích s ním porovnám. Tím získám informaci o tom, kde je větší g_c a kde g_p. V následujícím kroku provedu bitový posun doprava a dostanu se k nejsložitější části. Je důležité vybrat správné váhy, jakými budeme body násobit. Následnou sumu provedu jenom s body se kterými chceme pro daný g_c. Takto opakuji porovnávání a násobení váhami ještě 5x, kdy pro načtené 3 řádky dat jsem provedl 6 výpočtů g_p.

Ještě zpět k násobení správnými váhami. Algoritmus jsem si zjednodušil tak, že víme že $P = 8$ a $R = 1$. Pro první řádek víme, že budeme potřebovat mít váhy 1, 2 a 4. Jako nejrychlejší variantu mě napadlo mít předpřipravené 3 možnosti, kdy řádek budeme násobit buď 1,2,4,1,2,4,1,2, nebo 2,4,1,2,4,1,2,4 nebo 4,1,2,4,1,2,4,1. Pro následující řádky vytvořím násobkové váhy analogicky. Pak stačí vybrat správnou variantu a získám správně vynásobená data.

Algoritmus jsem následně upravil ještě tak, že když mám načtených 24 bodů, tak počítám hodnotu LBP pro 6 bodů. V úpravě jsem se snažil pro výpočet těchto 6ti bodů minimalizovat počet opakování cyklu. V minimalizované verzi těchto 6 bodů spočítám pouhými 3mi průchody, kdy každým průchodem spočítám 2 hodnoty LBP.



Obrázek 22: Ukázka výpočtu více hodnot LBP při jednom průchodu. Pro $P = 8$ a $R = 1$ nám u jak u 64-bitvého D registru, tak i u 128-bitového Q registru stačí na průchod a výpočet všech hodnot LBP 3 cykly.

Počet sloupců na řádku nemusí být nutně dělitelný číslem 8, proto všechny sloupce, na které nebyly vypočítány, musíme spočítat klasickým algoritmem uvedeným v kapitole 5.2.1 nebo jeho hardwarově akcelerovanou variantou uvedenou v kapitole 5.2.2.1.

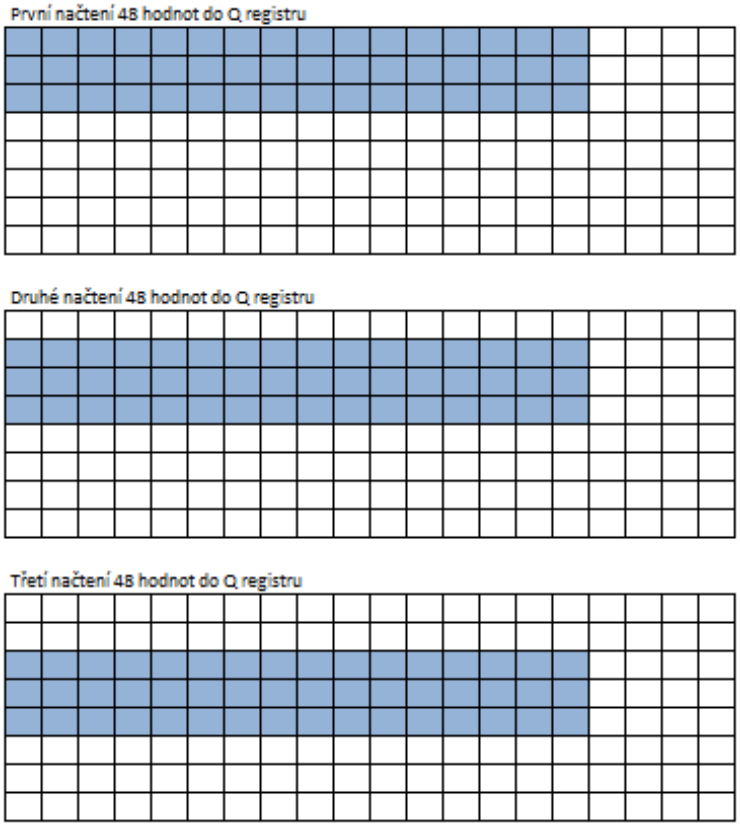
5.2.2.3 Verze 3 Tato varianta je vytvořena analogicky k variantě číslo 2 vytvořené v předchozí kapitole 5.2.2.2. Její hlavní úpravou je používání 128-bitové Q registru místo 64-bitového D registru. To znamená, že na řádku je uloženo 16 pixelů. Při dalším načtení tedy znovu načteno 6 pixelů z 48, tedy 1/8 oproti 1/4 v předchozím algoritmu.

Výpočet probíhá naprosto stejně jako u varianty číslo 2. Sloupce, na které nevyšlo při procházení po 16ti zkusím projít průchodem pomocí 8mi a následně dopočítáme klasickým algoritmem.

Když mám načtených všech 48 bodů, tak stejně jako u varianty číslo 2 nám pro průchod a vypočítání hodnot 14ti bodů stačí 3 cykly. Během těchto 3 cyklů vypočítám v prvním a druhém kroku 5 hodnot a v posledním kroku 4 hodnoty. Tímto způsobem jsem dosáhl toho, že místo abych porovnával, dělal bitový posun a násobil váhami 5x, tak to vše udělám během jednom cyklu.

5.2.2.4 Verze 4 Jedná se o variaci algoritmu v3, kdy po výpočtu 14tice se nebudeme pohybovat po ose x, ale po ose y s tím, že z předchozího průběhu si zapatuji 2 spodní řádky a načtu jenom jeden další řádek. Až vypočítám všechny řádky, tak se posuneme o

14 doprava a zahájíme nový průchod zhora dolů. Tímto způsobem jsem minimalizoval nutnost načítání do registrů na pouhý jeden řádek (toto se netýká, pokud máme $y = 0$, v daném případě musím načíst všechny 3 řádky, protože v daném případě jsem na horní části obrazu a potřebuji načíst informace o všech řádcích).



Obrázek 23: Ukázka průběhu výpočtu paralelního algoritmu pomocí NEON u verze 4. Body načítáme jako 3 řádky po 16ti bodech, celkem 48 bodů a z těchto načtených bodů vypočteme 14 hodnot LBP. Při načítání dalších 48 bodů načítáme po směru osy y.

5.2.3 Hardwarová akcelerace pomocí NEON assembleru

V této kapitole se budu věnovat, jak jsem hardwarově akceleroval algoritmus LBP pomocí NEON assembler instrukcí. Tímto způsobem jsem vytvořil 2 verze.

5.2.3.1 Verze 1 Tato verze je vytvořená analogicky k verzi 1 uvedená v kapitole 5.2.2.1, která je vytvořená podobně jako čistý C++ výpočet používán k výpočtu LBP. Všechny instrukce jsou shodné s hardwarově akcelerovanou verzí 1, jedná se o přepis instrukcí z C++ na assembler.

5.2.3.2 Verze 2 Tato verze byla vytvořena jako assembler verze k hardwarově akcelerované verzi 4 uvedené v kapitole 5.2.2.4. Oproti této verzi algoritmus načítá všechny řádky z RAM paměti. Je to způsobeno tím, že zatímco když používáme NEON C++ instrukce, tak jsem schopen používat C++ rozhodování a smyčky, zatímco v assembler verzi by jste tyto smyčky a ostatní logiku museli psát v assembleru. Proto došlo k zjednodušení algoritmu, čímž se vytratila ta vlastnost, že si algoritmus pamatuje 2 načtené řádky a načítá se už jenom jeden nový.

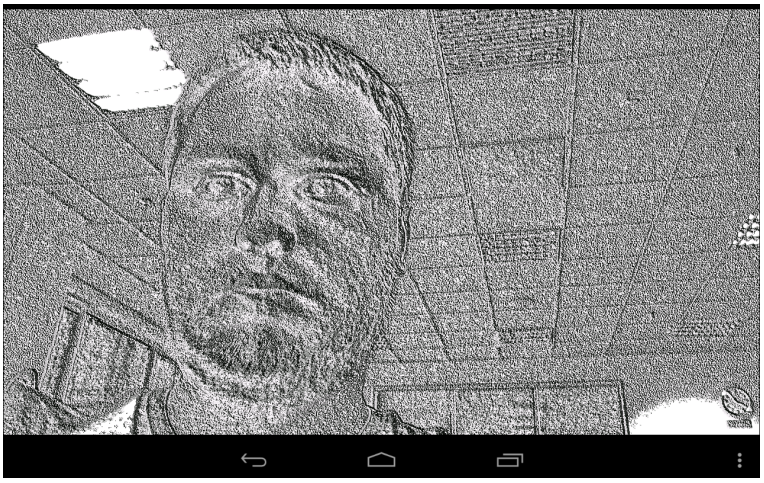
5.2.4 Srovnání výsledků

Na testovacím zařízení jsem pro implementované algoritmy naměřil časy uvedené v tabulce 4. Pro měření jsem ignoroval časy prvních 100 výpočtů a spočítal jsem čas potřebný k provedení následujících 500 zpracování algoritmu.

	LBP C	LBP NEON v1	LBP NEON v2	LBP NEON v3	LBP NEON v4
Celkem [ms]	252191	240283	212070	170197	128174
Průměr [ms]	504,382	480,566	424,14	340,394	256,348
Zrychlení [%]	0	5.31	15,91	32,51	49,18

Tabulka 4: Porovnání časů pro zpracování 500 snímků

Z výsledků v tabulkách 4 a 5 mi vyplývá, že u toho algoritmu se hardwarová akcelerace vyplatila. Skoro polovinu času pro výpočet jsem ušetřil využitím hardwarové akcelerace pomocí NEON C++ instrukcí a více než 60% času jsem ušetřil, když jsem použil NEON assembler instrukce.



Obrázek 24: Výstupní obraz LBP naimplementovaného v C++.

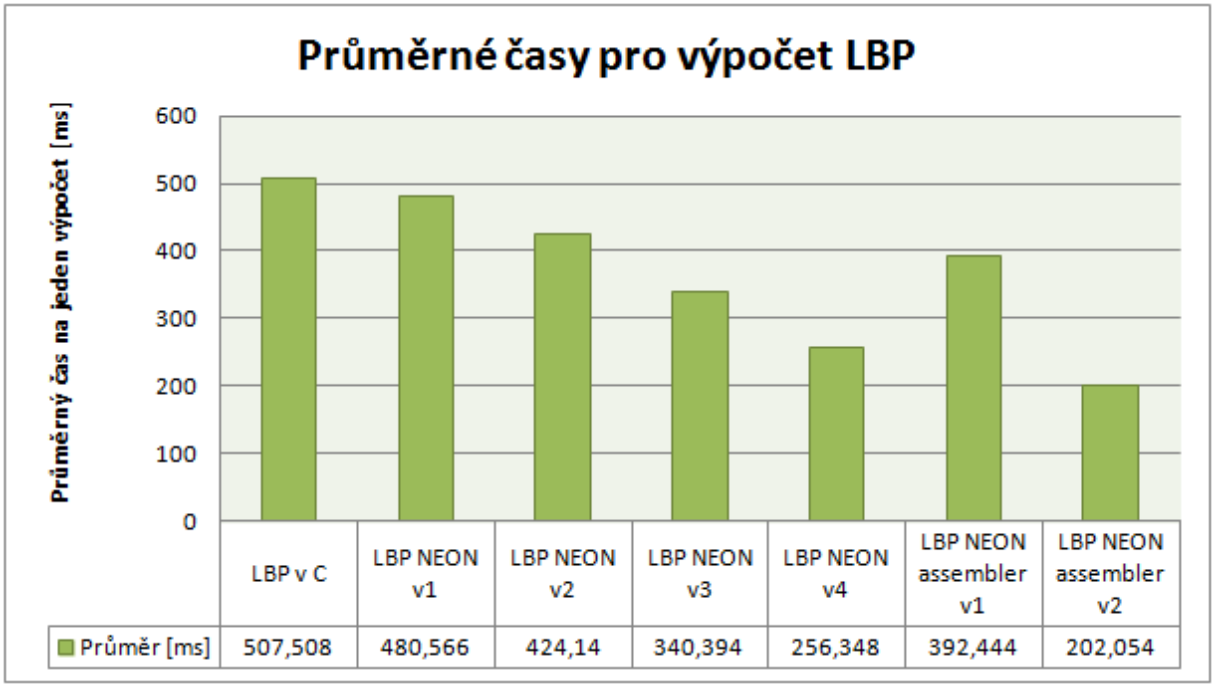
Nejrychlejším hardwarově akcelerovaným algoritmem pomocí NEON C++ instrukcí byla verze 4. Na výsledku verze 4 se dá poukázat na to, že načítání do registrů z paměti

RAM je pomalé, protože verze 3 a 4 se prakticky lišily jenom tím, že u verze 4 jsem si pamatoval 2 načtené řádky a načítal jsem pouze 1 nový. Toho jsem dosáhl tak, že jsem změnil typ posunu místo po sloupcích na posun po řádcích.

	LBP NEON assembler v1	LBP NEON assembler v2
Celkem [ms]	196222	101027
Průměr [ms]	392,444	202,054
Zrychlení [%]	22,67	60,19

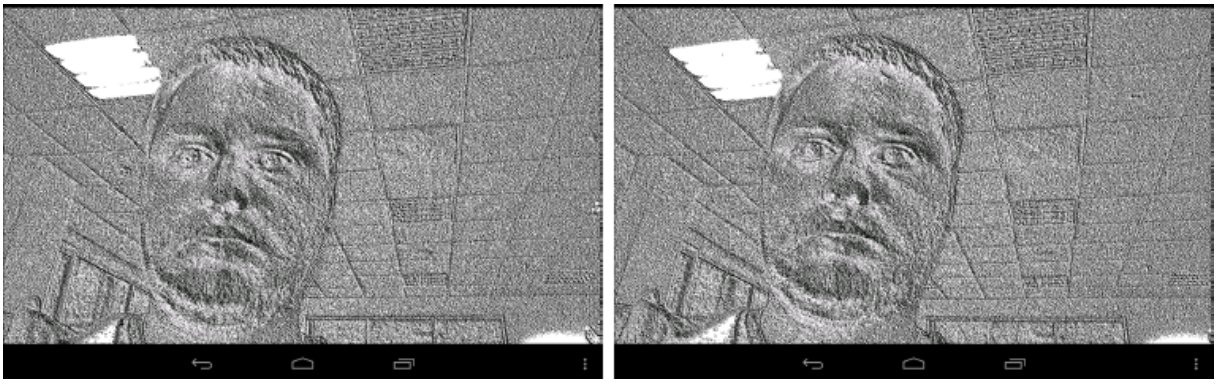
Tabulka 5: Porovnání časů pro zpracování 500 snímků pomocí NEON assembler instrukcí

Po nahlédnutí k časům uvedeným v tabulce 5 jsem usoudil, že implementace pomocí NEON assembler instrukcí je rychlejší, než verze z jaké vychází. Díky tomu se NEON assembler verze 2 stala nejrychleji počítaným algoritmem pro výpočet LBP. V daném případě se 60% času pro výpočet dalo ušetřit tím, že jsem použil NEON assembler instrukce.

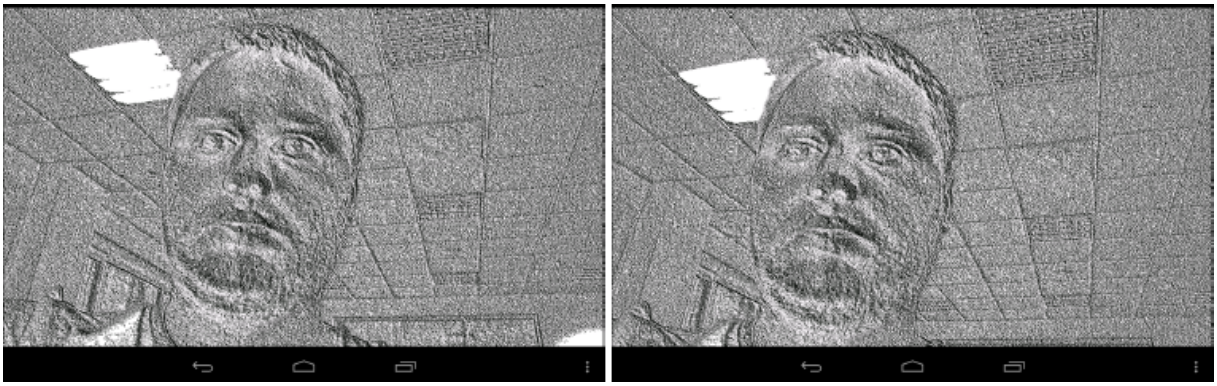


Obrázek 25: Graf průměrných časů pro výpočet LBP

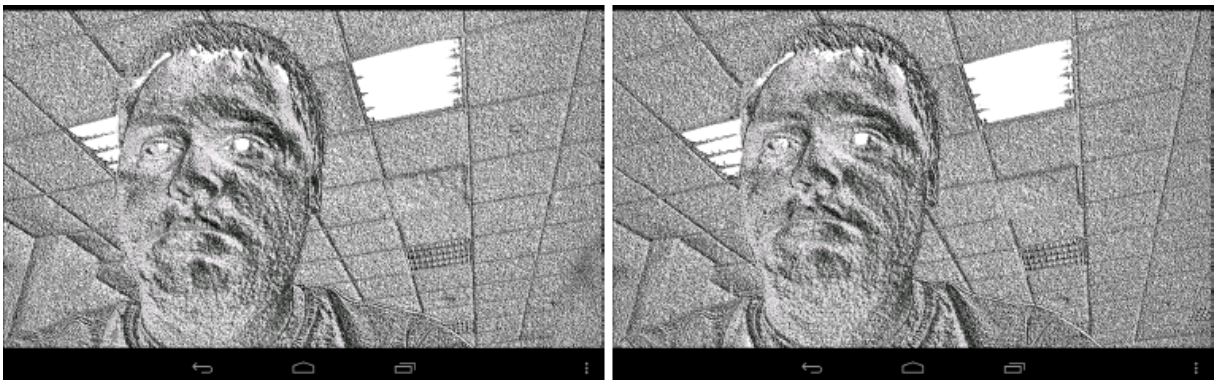
Z obrázků 24 jde vidět, jak vypadá výsledný obraz vytvořený pomocí jazyka C++ a na obrázcích 26 a 27 jde vidět, že jsem pro algoritmus hardwarově akcelerovaný pomocí NEON získal stejný výsledný obraz jako pro algoritmus napsaný čistě v C++. A na obrázku 28 jde vidět, že stejného výsledku jsem dosáhl i pro algoritmus LBP napsaný pomocí NEON assembler instrukcí.



Obrázek 26: Výstupní obraz LBP naimplementovaného v C++ a hardwarově akcelerován pomocí NEON verze 1 (vlevo) a verze 2 (vpravo).



Obrázek 27: Výstupní obraz LBP naimplementovaného v C++ a hardwarově akcelerován pomocí NEON verze 3 (vlevo) a verze 4 (vpravo).



Obrázek 28: Výstupní obraz LBP naimplementovaného v C++ a hardwarově akcelerován pomocí NEON assembler instrukcí verze 1 (vlevo) a verze 2 (vpravo).

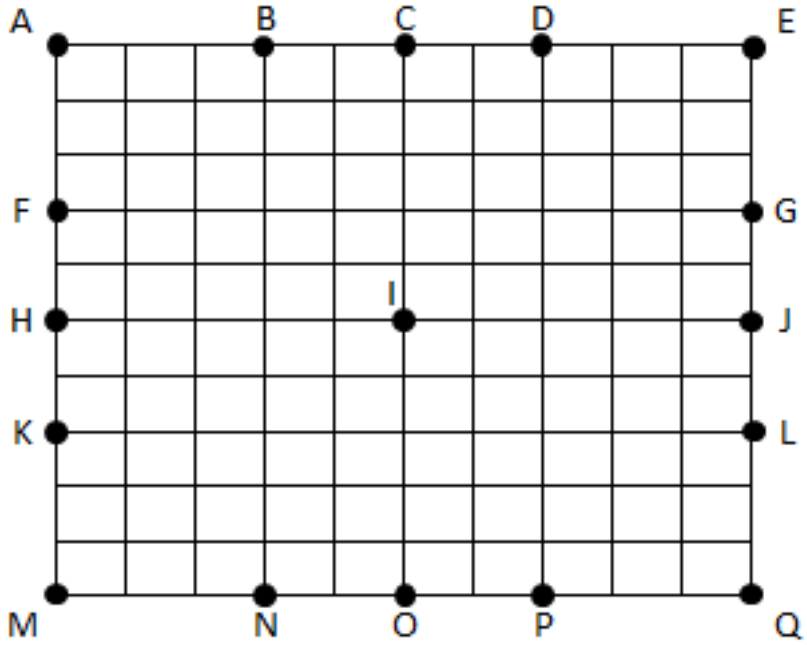
Kontrola funkčnosti mi ukázala, že výpočet pro všechny algoritmy je totožný a tedy algoritmy pro stejný vstup vypočítají totožný výstup.

5.3 Haarovy příznaky

V této kapitole popíšu způsob, jakým jsem implementoval výpočet Haarových příznaků.

Haarovy příznaky jsou výpočtově náročné, protože například při velikosti podokna 24x24 pixelů pro každé podokno počítáte 320 045 příznaků ze základní sady (2 hranové, 2 čárové a 1 diagonální). To nám vylučuje použití na reálném obraze v případě, že bych chtěl provádět výpočty opravdu svědomitě a s nárůstem o 1 pixel. Proto se v naší aplikaci budu věnovat části pro výpočet jednoho podokna a taky měřit celkový čas na výpočtu jednoho podokna. Toto podokno bude mít velikost 24x24 pixelů, velikost příznaku a posun příznaku budu zvětšovat o 2 pixely a celkově vypočítám 21 780 příznaků.

Pro výpočet základních příznaků (2 hranové, 2 čárové a 1 diagonální) jsem si zavedl 17 bodů, které budu používat při výpočtu hodnoty příznaku a těchto 17 bodů mi pomocí integrálního obrazu bude stačit na výpočet všech 5ti základních příznaků. Vybraných 17 bodů je ukázáno na obrázku 29.



Obrázek 29: 17 bodů nutných znát pro výpočet všech příznaků pro danou pozici podokna, velikost podokna, pozici příznaku a velikost příznaku. Pomocí těchto 17ti bodů a integrálního obrazu vypočítáme základní sadu Haarových příznaků (2 hranové, 2 čárové a 1 diagonální).

Pokud si vypočítám hodnotu celého příznaku, tak následně nebudu muset počítat hodnotu pro obě části zvlášť. Pro výpočet hodnoty celé oblasti využiji body uvedené na

obrázku 29 a získám vzorec, který vychází ze vzorce pro výpočet hodnoty z integrálního obrazu:

$$F_{total} = I(A) + I(Q) - I(E) - I(M) \tag{13}$$

Pro výpočet hodnot pro tmavší část Haarového příznaku využiji 5 vzorců odvozených stejným způsobem:

$$F_{black} = I(C) + I(Q) - I(E) - I(O) \tag{14}$$

$$F_{black} = I(H) + I(P) - I(J) - I(M) \tag{15}$$

$$F_{black} = I(A) + I(I) - I(C) - I(H) + I(I) + I(Q) - I(J) - I(O) \tag{16}$$

$$F_{black} = I(B) + I(P) - I(D) - I(N) \tag{17}$$

$$F_{black} = I(F) + I(L) - I(G) - I(K) \tag{18}$$

Pro výpočet bílé hodnoty využiji toho, že mám vypočítanou hodnotu černého příznaku a celkovou hodnotu. Bílou část tedy získám vzorcem:

$$F_{white} = F_{total} - F_{black} \tag{19}$$

A hodnotu příznaku pomocí vzorce:

$$F_{haar} = F_{white} - F_{black} = F_{total} - F_{black} - F_{black} = F_{total} - (2 * F_{black}) \tag{20}$$

Protože chci mít hodnotu příznaku normalizovanou, tak si pro každé podokno vypočítám směrodatnou odchylku podle vzorce:

$$\sigma = \sqrt{\frac{I^2(w,h)}{wh} - \left(\frac{I(w,h)}{wh}\right)^2} \tag{21}$$

A normalizovanou hodnotu získám:

$$F_{normalized} = \frac{F_{haar}}{\sigma} \tag{22}$$

Tento výpočet budu tedy implementovat v jednotlivých algoritmech.

5.3.1 Implementace v C++

Pro implementaci budu potřebovat 7 vnořených smyček. 1 smyčka pro změnu velikosti podokna, 2 smyčky pro posun podokna po obraze.

Nyní bude hlavní část a to výpočet příznaků pro jednotlivá podokna. Zde budu mít další 4 smyčky, 2 smyčky budou pro změnu velikosti příznaku a 2 smyčky budou pro posun příznaku po obrázku.

Těchto 7 smyček vylučuje použití v aplikacích pro zpracování obrazu v reálném času, pokud budu chtít počítat opravdu poctivě. U posledních 4 smyček jsem se rozhodl pro dvounásobný krok. Pro použití v aplikaci zpracovávající obraz v reálném čase bych ale požadavky musel mnohem více snížit, hlavně by bylo potřeba snížit počet podoken, nejlépe výpočet těchto podoken zrušit a počítat rovnou pro celý obraz.

Každý vypočítaný příznak si ukládáme do listu a na konci nám funkce vrátí tento list.

5.3.2 Hardwarová akcelerace pomocí NEON C++

Na začátku bylo důležité zamyslet se nad datovými typy, které použiji při hardwarové akceleraci. Vzhledem k tomu, že výstupní hodnota má datový typ float a NEON neumí provádět operace mezi různými datovými typy, tak vstupy již musíme načíst jako datový typ float32_t. Následně si ještě načtu směrodatnou odchylku v invertované podobě, protože NEON neumí dělit, takže musíme použít invertovanou hodnotu a násobit.

Vzhledem k tomu, že počet Q registrů máme omezený a pokud bych použil D registr, tak by ke zrychlení s určitostí nedošlo, tak jsem se rozhodl výpočet rozdělit na 2 části. V první části načtu 9 bodů potřebných pro výpočet hranových a diagonálního příznaku. Po tom co tyto příznaky vypočítám, tak do 8mi registrů načtu hodnoty 8mi bodů potřebných pro výpočet čárových příznaků.

5.3.3 Hardwarová akcelerace pomocí NEON assembleru

Jde o přepis algoritmu z kapitoly 5.3.2 z NEON C++ instrukcí na assembler instrukce s pár změnami. Protože výpočet mám rozdělený na 2 části, tak si hodnotu invertované směrodatné odchylky načtu v každé části a hodnotu celkové sumy příznaku si taky počítám v každé části. Ostatní funkčnost a výpočty zůstávají totožné.

Znovu jsem narazil na bug kompilátoru [60]. V tomto případě se mi podařilo zjistit, že asm akceptuje maximálně 13 vstupů/výstupů. Ale pro správný výpočet potřebuji 14 vstupů/výstupů. Menší úpravou výpočtu se mi povedlo snížit potřebný počet vstupů o jeden, takže mi začalo stačit 13 vstupů. Následně jsem zjistil to, že budu potřebovat výstup navíc a to pro uložení hodnoty pro celý příznak, aby se dal znovu načíst v druhém výpočtu. Jako řešení jsem se rozhodl, že výsledek si uložím do jednoho ze vstupů.

5.3.4 Srovnání výsledků

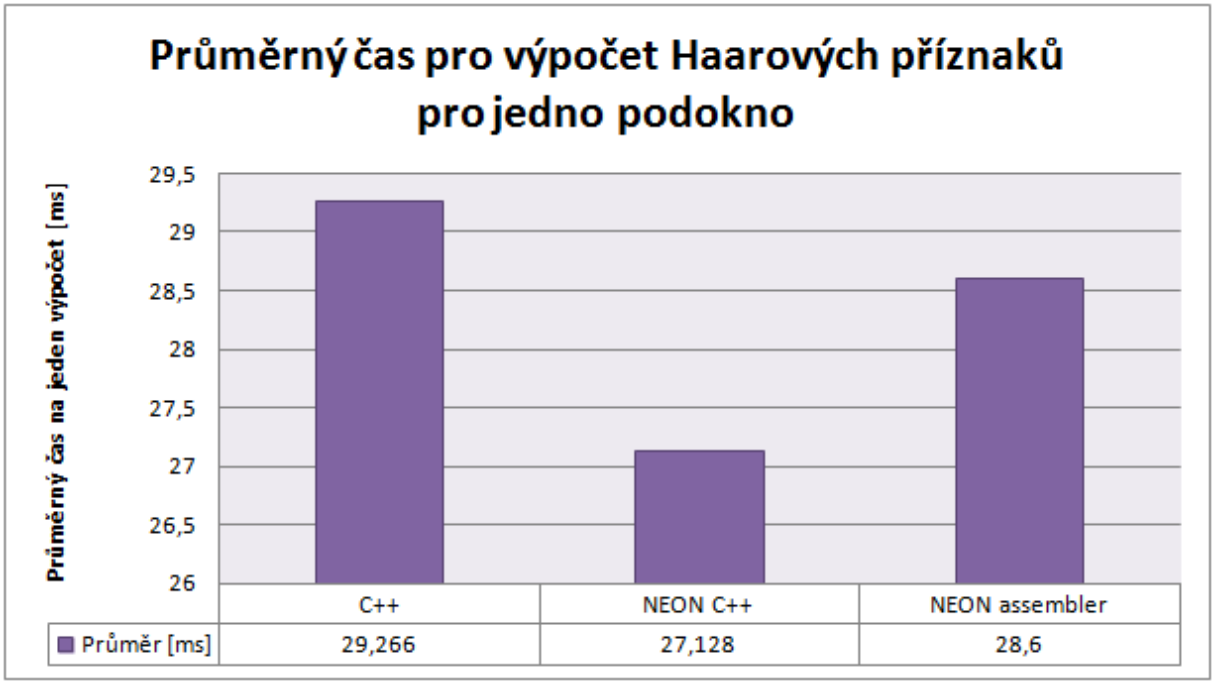
Na testovacím zařízení Nexus 7 jsem pro implementované algoritmy naměřil časy uvedené v tabulce 6. Měření jsem prováděl pro podokna a jako vždy jsem ignoroval prvních 100 podoken a měřil jsem čas pro dalších 500 podoken.

	C++	NEON C++	NEON assembler
Celkem [ms]	14633	13564	14300
Průměr [ms]	29,266	27,128	28,6
Zrychlení [%]	0	7,31	2,28

Tabulka 6: Porovnání časů pro zpracování 500 snímků pomocí NEON assembler instrukcí

U dosažených výsledků lze vidět, že v tomto případě je výpočet pomocí NEON assembler instrukcí pomalejší, než výpočet pomocí NEON C++ instrukcí. Je to způsobeno tím, že výpočet máme rozdělený na 2 části a tak dojde k přesunu informací do čipu pro výpočet NEON instrukcí, k návratu z něho a znovu k přesunu dat oproti verzi napsané pomocí NEON C++ instrukcí, kdy dojde jenom k přesunu části dat a pokračování ve

výpočtu. I toto zpoždění nám ovlivní výslede tak, že verze hardwarově akcelerovaná pomocí NEON C++ instrukcí je nejrychlejší způsob výpočtu Haarových příznaků.



Obrázek 30: Graf průměrných časů pro výpočet Haarových příznaků pro jedno podokno

Funkčnost jsem ověřil tak, že na obraze porovnám vypočítané hodnoty příznaku pro všechny algoritmy. Tímto způsobem jsem si ověřil, že všemi algoritmy dostanu stejné výsledky až občasné výjimky. Občas dojde k rozdílu hodnoty, kdy pro hardwarově akcelerované algoritmy získám stejný výsledek, ale pro algoritmus napsaný v C++ získáme výsledek, který se liší o desetinu procenta hodnoty. Tento rozdíl může být způsobený rozdílným použitím datových typů, kdy pro hardwarově akcelerované algoritmy musím používat datový typ s plovoucí desetinnou čárkou, zatímco pro algoritmus napsaný čistě v C++ používám pro část výpočtu celočíselný datový typ a až následně datový typ s plovoucí desetinnou čárkou.

6 Závěr

Práce se zabývala hardwarovou akcelerací vybraných algoritmů. Cílem práce bylo seznámit se s algoritmy pro hledaní vzorů pomocí hrubé síly, Lokální binární vzory, Haarovy příznaky a následně vytvořit hardwarově akcelerované varianty těchto algoritmů. Tyto hardwarově akcelerované algoritmy měly být funkční na mobilní platformě Nvidia.

Z toho důvodu jsem jako nejlepší možnost pro hardwarovou akceleraci vybral technologii Advanced SIMD extension - NEON. NEON je rozšíření instrukční sady ARMv7 procesorů umožňující SIMD výpočty na mobilních zařízeních. Instrukce NEON můžeme psát buď jako C++ funkce, nebo jako assembler instrukce. V této práci jsem vyzkoušel oba způsoby a porovnával jsem tak rychlost algoritmu napsaného v C++, algoritmem hardwarově akcelerovaného pomocí NEON C++ instrukcí a algoritmem hardwarově akcelerovaného pomocí NEON assembleru.

Z výsledků měření jsem odvodil, že použití NEON assembleru dokáže urychlit již hardwarově akcelerovaný algoritmus pomocí NEON C++ instrukcí. Toho jsem docílil u 2 z 3 pokusů. U třetího pokusu nedošlo ke zlepšení u NEON assembler instrukcí z důvodu omezení oproti použití NEON C++ instrukcí.

U 2 z 3 pokusů jsem taky dokázal, že hardwarově akcelerovaný algoritmus je rychlejší, než verze napsaná v C++. U třetího algoritmu, kde ke zrychlení nedošlo, jsem si ověřili, že SIMD výpočty nejsou vhodné pro algoritmy, kde aktuálně počítaná hodnota i z části závisí na hodnotě předchozí.

Pro hardwarově akcelerované algoritmy jsem vytvořil více verzí, abych předvedl rychlostní rozdíly u datových typů nebo rozdíly mezi NEON registry.

7 Reference

[1] *Konstantinos G. Derpanis* [online]. [cit. 2014-4-26]. Integral Image-based Representations paper. Dostupné z WWW: <http://herkules.oulu.fi/isbn9514270762/isbn9514270762.pdf>

[2] *R. Brunelli* [2009]. Template Matching Techniques in Computer Vision: Theory and Practice, Wiley.

[3] *T. Ojala, M. Pietikäinen, and D. Harwood* [1994]. Performance evaluation of texture measures with classification based on Kullback discrimination of distributions, Proceedings of the 12th IAPR International Conference on Pattern Recognition (ICPR 1994), vol. 1.

[4] *T. Ojala, M. Pietikäinen, and D. Harwood* [1996]. A Comparative Study of Texture Measures with Classification Based on Feature Distributions, Pattern Recognition.

[5] *Mäenpää, T* [online]. [cit. 2014-4-26]. The Local binary pattern approach to texture analysis: extensions and applications. Dostupné z WWW: <http://herkules.oulu.fi/isbn9514270762/isbn9514270762.pdf>

[6] *T. Ojala, M. Pietikäinen, M.; Mäenpää, T* [1996]. Multiresolution Gray-Scale and Rotation Invariant Texture Classification with Local Binary Patterns, 2002. IEEE Trans. Pattern Anal. Mach. Intell. 24(7): strany 971-987

[7] *Viola and Jones* [2001]. Rapid object detection using a boosted cascade of simple features, Computer Vision and Pattern Recognition, 2001 Dostupné z WWW: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.10.6807&rep=rep1&type=pdf>

[8] *Gerónimo David* [online]. [cit. 2014-4-26]. Haar-likeFeatures and Integral ImageRepresentation. Barcelona : Universitat Autonoma de Barcelona, 2009. 46 s. Výuková prezentace. Universitat Autonoma de Barcelona. Dostupné z WWW: <www.cvc.uab.es/adas>

[9] *NVIDIA* NVIDIA Rolls out "Tegra"Processors [online]. [cit. 2008-06-02]. Dostupné z WWW: <http://www.techtree.com/India/News/Nvidia_Rolls_out_Tegra_Processors/551-89833-581.html>

[10] *Nvidia Tegra2* Benefits of Multi core CPUs in Mobile Devices [online]. [cit. 2008-06-02]. Dostupné z WWW: <http://www.nvidia.com/content/PDF/tegra_white_papers/Benefits-of-Multi-core-CPUs-in-Mobile-Devices_Ver1.2.pdf>

[11] *Peter Clarke, EE Times* [online]. [cit. 2014-4-26] Audi selects Tegra processor for infotainment, dashboard. Dostupné z WWW: <http://www.eetimes.com/document.asp?doc_id=1260984>

[12] *Nvidia Tegra 4* [online]. [cit. 2014-4-26]. Dostupné z WWW: <http://www.nvidia.com/object/tegra-4-processor.html>

[13] *Nvidia Tegra 4* NVIDIA Tegra 4 Family GPU Architecture [online]. [cit. 2014-4-26]. Dostupné z WWW: <http://www.nvidia.com/docs/IO//116757/Tegra_4_GPU_Whitepaper_FINALv2.pdf>

[14] *NVIDIA® Tegra® mobile processors - Tegra 2* [online]. [cit. 2008-06-02]. Dostupné z WWW: <http://www.nvidia.com/object/tegra-superchip.html>

[15] *Nvidia Tegra 3* [online]. [cit. 2014-4-26]. Variable SMP – A Multi-Core CPU Architecture for Low Power and High Performance. Dostupné z WWW: <http://www.nvidia.com/content/PDF/tegra_white_papers/tegra-whitepaper-0911b.pdf>

[16] *NVIDIA® Tegra® mobile processors - Tegra 3* [online]. [cit. 2014-4-26]. Dostupné z WWW: <http://www.nvidia.com/object/tegra-3-processor.html>

[17] *Nvidia Tegra K1* [online]. [cit. 2014-4-26]. Dostupné z WWW: <http://www.nvidia.com/object/tegra-k1-processor.html>

[18] *Nvidia Tegra K1* [online]. A New Era in Mobile Computing [cit. 2014-4-26]. Dostupné z WWW: <http://www.nvidia.co.uk/content/PDF/tegra_white_papers/tegra-K1-whitepaper.pdf>

[19] *ARM aims son of Thumb at uCs, ASSPs, SoCs* [online]. [cit. 2014-4-26]. Dostupné z WWW: <http://archive.today/1Lao>

[20] *RM1156T2F-S architecture with Thumb-2 core technology* [online]. [cit. 2014-4-26]. Dostupné z WWW: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0290g/I1005458.html>

[21] *ARM Processor Architecture* [online]. [cit. 2014-4-26] Dostupné z WWW: <http://www.arm.com/products/processors/instruction-set-architectures/index.php>

[22] *VFP directives and vector notation* [online]. [cit. 2014-4-26]. Dostupné z WWW: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0204j/Chdehgeh.html>

[23] *Floating Point* [online]. [cit. 2014-4-26]. Dostupné z WWW: <http://www.arm.com/products/processors/technologies/vector-floating-point.php>

[24] *Differences between ARM Cortex-A8 and Cortex-A9* [online]. [cit. 2014-4-26]. Dostupné z WWW: <http://www.shervinemami.info/armAssembly.html#cortex-a9>

[25] *Cortex-R4 Processor* [online]. [cit. 2014-4-26]. Dostupné z WWW: <http://www.arm.com/products/processors/cortex-r/cortex-r4.php>

[26] *About the Cortex-A9 NEON MPE* [online]. [cit. 2014-4-26]. Dostupné z WWW: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0409f/Chdceejc.html>

[27] *Vector data types* [online]. [cit. 2014-4-26]. Dostupné z WWW: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0491c/BABDEAGJ.html>

[28] *Intrinsics* [online]. [cit. 2014-4-26]. Dostupné z WWW: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0491c/BABDEAGJ.html>

[29] *Loads of a single vector or lane* [online]. [cit. 2014-4-26]. Dostupné z WWW: <http://infocenter.arm.com/help/topic/com.arm.doc.dui0491c/BABDCGGF.html>

[30] *Store a single vector or lane* [online]. [cit. 2014-4-26]. Dostupné z WWW: <http://infocenter.arm.com/help/topic/com.arm.doc.dui0491c/BEHHCHAE.html>

[31] *Addition* [online]. [cit. 2014-4-26]. Dostupné z WWW: <http://infocenter.arm.com/help/topic/com.arm.doc.dui0491c/BABDFJCI.html>

[32] *Subtraction* [online]. [cit. 2014-4-26]. Dostupné z WWW: <http://infocenter.arm.com/help/topic/com.arm.doc.dui0491c/BABIBDGG.html>

[33] *Subtraction* [online]. [cit. 2014-4-26]. Dostupné z WWW: <http://infocenter.arm.com/help/topic/com.arm.doc.dui0491c/BABEDJFB.html>

[34] *Multiplication* [online]. [cit. 2014-4-26]. Dostupné z WWW: <http://infocenter.arm.com/help/topic/com.arm.doc.dui0491c/BABDEAGJ.html>

[35] *Shifts by a constant* [online]. [cit. 2014-4-26]. Dostupné z WWW: <http://infocenter.arm.com/help/topic/com.arm.doc.dui0491c/BABIBBJG.html>

[36] *Shifts by signed variable* [online]. [cit. 2014-4-26]. Dostupné z WWW: <http://infocenter.arm.com/help/topic/com.arm.doc.dui0491c/BABDFGBJ.html>

[37] *Comparison* [online]. [cit. 2014-4-26]. Dostupné z WWW: <http://infocenter.arm.com/help/topic/com.arm.doc.dui0491c/BABGDDDH.html>

[38] *Android source* [online]. 2009 [cit. 2014-3-21]. About Android Dostupné z WWW: <http://source.android.com/about/index.html>.

[39] *Android developer* [online]. 2009 [cit. 2014-3-21]. Publishing on Android market. Dostupné z WWW: <http://developer.android.com/guide/publishing/publishing.html>

[40] *ELGIN B. Business Week* [online]. August 2005 [cit. 2014-3-20]. Google Buys Android for Its Mobile Arsenal. Dostupné z WWW: <http://www.businessweek.com/technology/content/aug2005/tc20050817_0949_tc024.htm>

[41] *Android developer* [online]. 2009 [cit. 2014-3-21]. What is Android. Dostupné z WWW: <http://developer.android.com/guide/basics/what-is-android.html>

[42] *Open Headset Aliance* [online]. 2009 [cit. 2014-3-21]. Dostupné z WWW: <http://www.openhandsetalliance.com/>

[43] *Open Headset Aliance* [online]. November 2007 [cit. 2014-3-20]. Industry Leaders Announce Open Platform for Mobile Devices. Dostupné z WWW: <http://www.openhandsetalliance.com/press_110507.html>

[44] *Open Headset Aliance* [online]. November 2007 [cit. 2014-3-20]. Open Handset Alliance Releases Android SDK. Dostupné z WWW: <http://www.openhandsetalliance.com/press_111207.html>

[45] MORRIL, *Android-developers Blogspot* [online]. September 2008 [cit. 2014-4-21]. Announcing the Android 1.0 SDK, release 1. Dostupné z WWW: <http://android-developers.blogspot.com/2008/09/announcing-android-10-sdk-release-1.html>

[46] *Android developer* [online]. [cit. 2014-4-21]. Application Fundamentals. Dostupné z WWW: <http://developer.android.com/guide/topics/fundamentals.html>

[47] *Android developer* [online]. [cit. 2014-4-21]. Manifest XML. Dostupné z WWW: <http://developer.android.com/guide/topics/manifest/manifest-intro.html>

[48] *Android developer* [online]. [cit. 2014-4-21]. Security. Dostupné z WWW: <http://developer.android.com/guide/topics/security/security.html>

[49] *Android developer* [online]. [cit. 2014-4-21]. Profiling with Traceview and dmtracedump. Dostupné z WWW: <http://developer.android.com/guide/developing/debugging/debugging-tracing.html>

[50] *Android developer* [online]. [cit. 2014-4-21]. Graphics. Dostupné z WWW: <http://developer.android.com/guide/topics/graphics/index.html>

[51] *Android developer, NDK* [online]. [cit. 2014-4-26]. Dostupné z WWW: <http://developer.android.com/sdk/ndk/index.html>

[52] *Mind the robot, NDK setup step by step* [online]. [cit. 2014-4-26]. Dostupné z WWW: <http://mindtherobot.com/blog/452/android-beginners-ndk-setup-step-by-step/>

[53] *OpenCV* [online]. [cit. 2014-4-26]. About OpenCV. Dostupné z WWW: <http://opencv.org/about.html>

[54] *OpenCV* [online]. [cit. 2014-4-26]. Cuda GPU port. Dostupné z WWW: <http://opencv.org/platforms/cuda.html>

[55] *OpenCV* [online]. [cit. 2014-4-26]. OpenCL Announcement. Dostupné z WWW: <http://opencv.org/opencv-v2-4-3rc-is-under-way.html>

[56] *OpenCV* [online]. [cit. 2014-4-26]. OpenCL-accelerated Computer Vision API Reference. Dostupné z WWW: <http://docs.opencv.org/modules/ocl/doc/ocl.html>

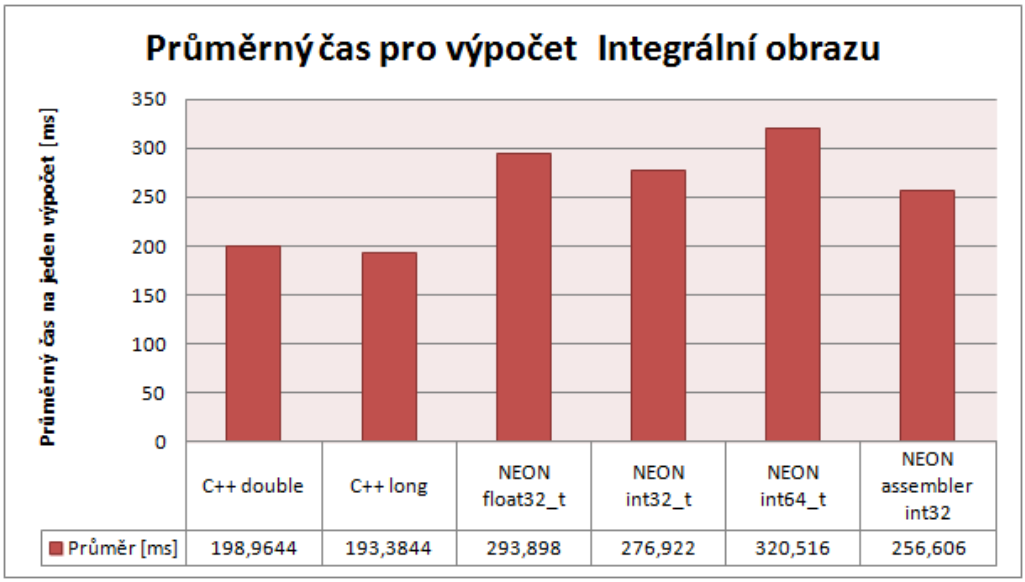
[57] *Android Developers* [online]. [cit. 2014-4-26]. Android NDK & ARM NEON instruction set extension support. Dostupné z WWW: <http://www.kandroid.org/ndk/docs/CPU-ARM-NEON.html>

[58] *Assembler* [online]. [cit. 2014-4-26]. Introducing NEON Development Article. Dostupné z WWW: <http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html>

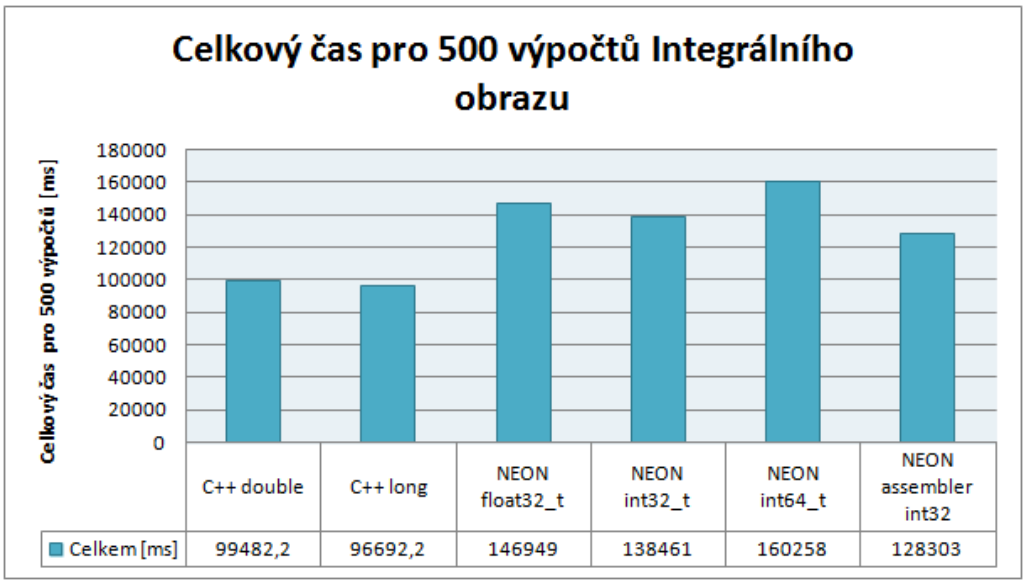
[59] *Assembler* [online]. [cit. 2014-4-26]. Introducing NEON Development Article. Dostupné z WWW: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dht0002a/CACGDCAH.html>

[60] *Bug 13850* [online]. [cit. 2014-4-26]. Can't find a register in class 'GENERAL_REGS' while reloading 'asm'. Dostupné z WWW: <http://gcc.gnu.org/bugzilla/show_bug.cgi?id=13850>

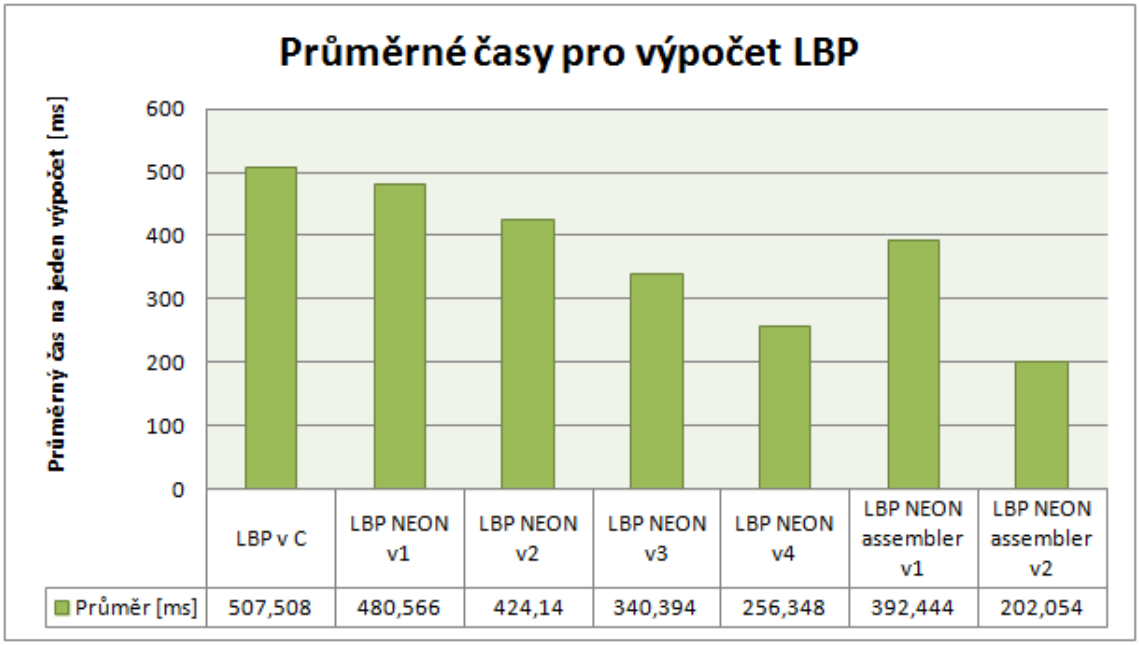
A Grafy



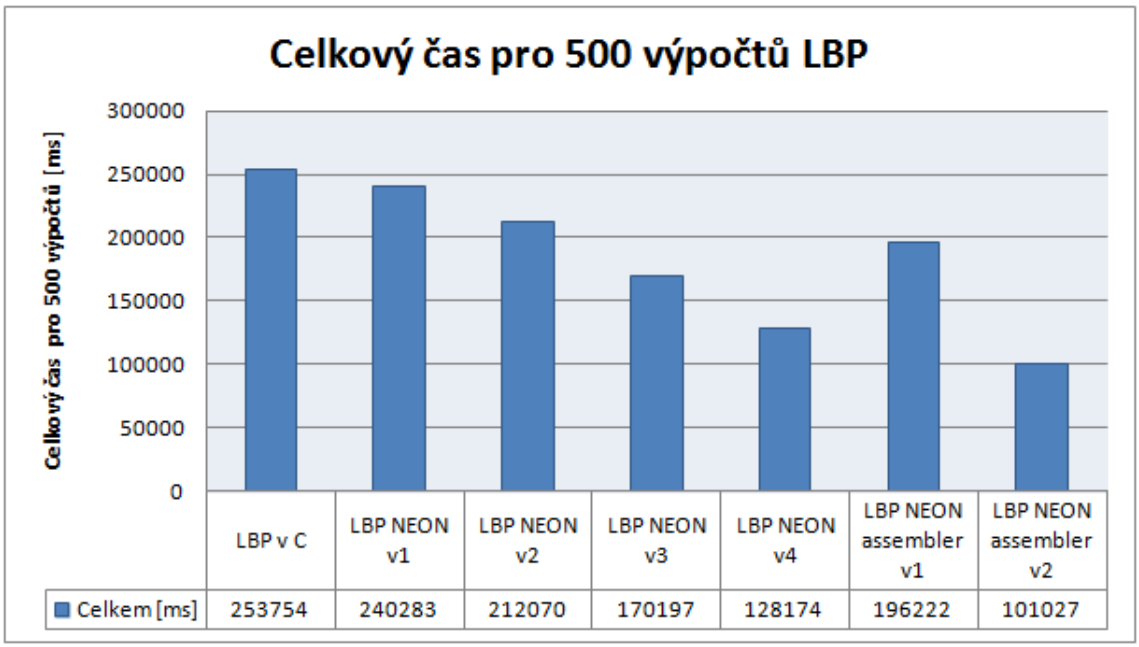
Obrázek 31: Graf průměrných časů pro výpočet Integrálního obrazu



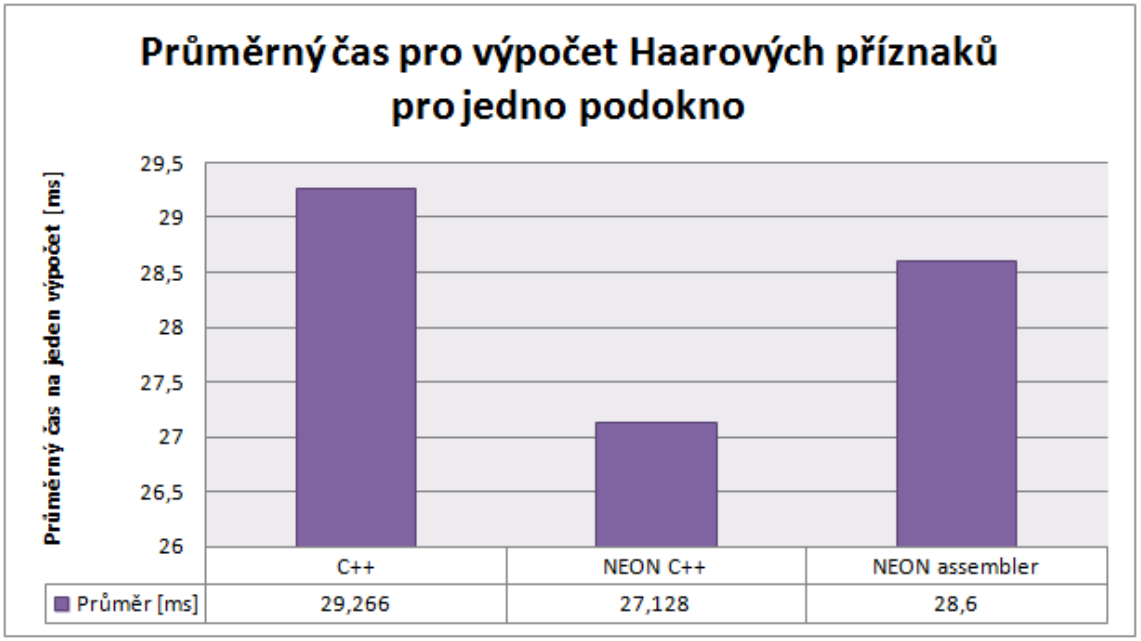
Obrázek 32: Graf celkových časů pro výpočet Integrální obrazu pro 500 výpočtů algoritmu



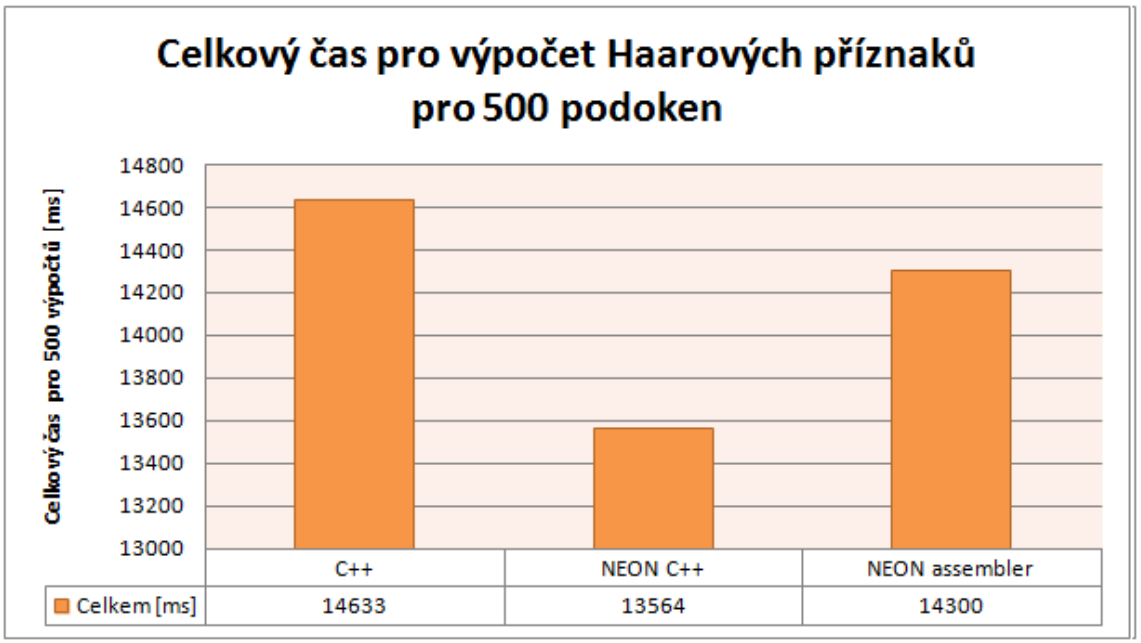
Obrázek 33: Graf průměrných časů pro výpočet LBP



Obrázek 34: Graf celkových časů pro výpočet LBP pro 500 výpočtů algoritmu



Obrázek 35: Graf průměrných časů pro výpočet Haarových příznaků pro jedno podokno



Obrázek 36: Graf celkových časů pro výpočet Haarových příznaků pro 500 podoken